THÈSE

présentée à

l'Université Paris 7 – Denis Diderot

pour obtenir le titre de

Docteur en Informatique

A Shared Memory Poetics

soutenue par

JADE ALGLAVE

le 26 Novembre 2010

JURY

Président	Ahmed Bouajjani
Rapporteurs	Gérard Boudol Maurice Herlihy
Examinateurs	Peter O'Hearn Peter Sewell
Directeurs	Jean-Jacques Lévy Luc Maranget

Contents

I Preamble **11**

1 Introduction **13**
- 1.1 Context . . . 13
  - 1.1.1 Early Days (1979 – 1995) . . . 13
  - 1.1.2 Recent Days (2002 – 2010) . . . 19
- 1.2 Contribution . . . 20
  - 1.2.1 A Generic Framework . . . 20
  - 1.2.2 A Testing Tool . . . 20
  - 1.2.3 Synchronisation . . . 21
  - 1.2.4 Remainder . . . 21

2 Preliminaries **23**
- 2.1 Relations . . . 23
  - 2.1.1 Basic Definitions . . . 23
  - 2.1.2 Orders . . . 24
- 2.2 Linear Extension . . . 25
- 2.3 A Key Lemma . . . 25
  - 2.3.1 Hexa Relation . . . 26
  - 2.3.2 Proof of the Result . . . 27

II A Generic Framework For Weak Memory Models **29**

3 A Generic Framework **33**
- 3.1 Basic Objects . . . 33
  - 3.1.1 Events and Program Order . . . 35
- 3.2 Execution Witnesses . . . 35
  - 3.2.1 Read-From Map . . . 38
  - 3.2.2 Write Serialisation . . . 38
  - 3.2.3 From-Read Map . . . 39
  - 3.2.4 All Together . . . 39
- 3.3 Global Happens-Before . . . 40
  - 3.3.1 Globality . . . 41

3.3.2 Preserved Program Order 42
3.3.3 Barriers Constraints 42
3.3.4 Architectures 43
3.3.5 Examples 44
3.4 Validity of an Execution 45
3.4.1 Uniprocessor Behaviour 45
3.4.2 Thin Air 48
3.4.3 Validity 49
3.5 Comparing Architectures 50
3.5.1 Validity Is Decreasing 50
3.5.2 Monotonicity of Validity 50

4 Classical Models 53
4.1 Implementing an Architecture 53
4.1.1 Building an Execution Witness From an Order 54
4.1.2 Sketch of Proof 57
4.2 A Hierarchy of Classical Models 59
4.2.1 Sequential Consistency (SC) 60
4.2.2 The Sparc Hierarchy 61
4.2.3 Alpha 65
4.2.4 RMO and Alpha Are Incomparable 67

5 Related Work 71
5.1 Generic Models 71
5.2 Global-Time vs. View Orders 72
5.3 Axiomatic vs. Operational 72
5.4 Characterisation of Behaviours 73
5.5 Memory Models As Program Transformations 73

III Testing Weak Memory Models 75

6 Relaxations 79
6.1 A Brief Glance at the Power Documentation 79
6.1.1 Axioms of Our Model 79
6.1.2 Store Buffering 80
6.1.3 Load-Load Pairs 81
6.1.4 Load-Store Pairs 81
6.1.5 Barriers 82
6.2 Candidate Relaxations 85
6.2.1 Communication Candidate Relaxations 85
6.2.2 Program Order Candidate Relaxations 85
6.2.3 Barriers Candidate Relaxations 86
6.2.4 Dependencies Candidate Relaxations 87

6.2.5 Composite Candidate Relaxations 88
6.3 A Preliminary Power Model 89

7 Diy, A Testing Tool 91
7.1 Litmus Tests 92
7.1.1 Highlighting Relaxations 92
7.1.2 Exercising One Relaxation at a Time 92
7.2 Cycles as Specifications of Litmus Tests 93
7.2.1 Automatic Test Generation 93
7.2.2 Cycles Generation 95
7.3 Code Generation 96
7.3.1 Algorithm 96
7.3.2 Example 97
7.4 A First Testing Example: x86-TSO 98
7.4.1 A Guided Diy Run 99
7.4.2 Configuration Files 101

8 A Power Model 103
8.1 The Phat Experiment 104
8.1.1 Relaxations Observed on `squale`, `vargas` and `hpcx` . . 104
8.1.2 Safe Relaxations 104
8.2 Overview of Our Model 106
8.2.1 Additional Formalism 106
8.2.2 Description of the Model 110
8.3 Discussion of Our Model 118

9 Related Work 121

IV Synchronisation in Weak Memory Models 123

10 Synchronisation 127
10.1 Covering and well-founded relations 127
10.1.1 Covering relations 127
10.1.2 Well-founded relations 128
10.2 DRF guarantee 129
10.2.1 Competing accesses 129
10.2.2 Synchronising competing accesses in a weak execution 130
10.2.3 DRF guarantee 131
10.3 Lock-free guarantee 132
10.3.1 Fragile pairs 132
10.3.2 Synchronising fragile pairs in a weak execution 133
10.3.3 Application to the Semantics of Barriers 134
10.3.4 Lock-free guarantee 137

10.4 Synchronisation idioms 138
10.4.1 Atomicity 139
10.4.2 Locks 139
10.4.3 Lock-free synchronisation 145

11 Stability **149**
11.1 Minimal cycles 149
11.1.1 Violations 149
11.1.2 Covering the minimal violations 151
11.1.3 Critical cycles 151
11.1.4 A characterisation of the minimal cycles 154
11.2 Stability from any architecture to SC 155

12 Related Work **159**

V Conclusion and Perspectives **163**

13 Conclusion **165**
13.1 Divining Chicken Entrails 165
13.1.1 Reading the Documentations 165
13.1.2 Abstract Models 166
13.1.3 Simple Formal Models 166
13.1.4 The Preserved Program Order Quest 167
13.1.5 Strong Programming Disciplines 167
13.2 A Reading Frame For Weak Memory Models 168
13.2.1 A Common Prism 168
13.2.2 Tests as Specifications 168

14 Perspectives **169**
14.1 Automatisation 169
14.2 Formalisation of Diy 169
14.3 Other Models And Paradigms 170
14.4 Testing Semantics For Weak Memory Models 170
14.5 Logics For Weak Memory Models 171
14.6 Partial Orders As a Model of Concurrency 172

VI Appendix **181**

A Uniprocessor Equivalences **183**
A.1 Some Handy Lemmas 183
A.2 (*Uni2*) and (*Uni3*) Are Equivalent 184
A.3 (*Uni1*) and (*Uni2*) Are Equivalent 184

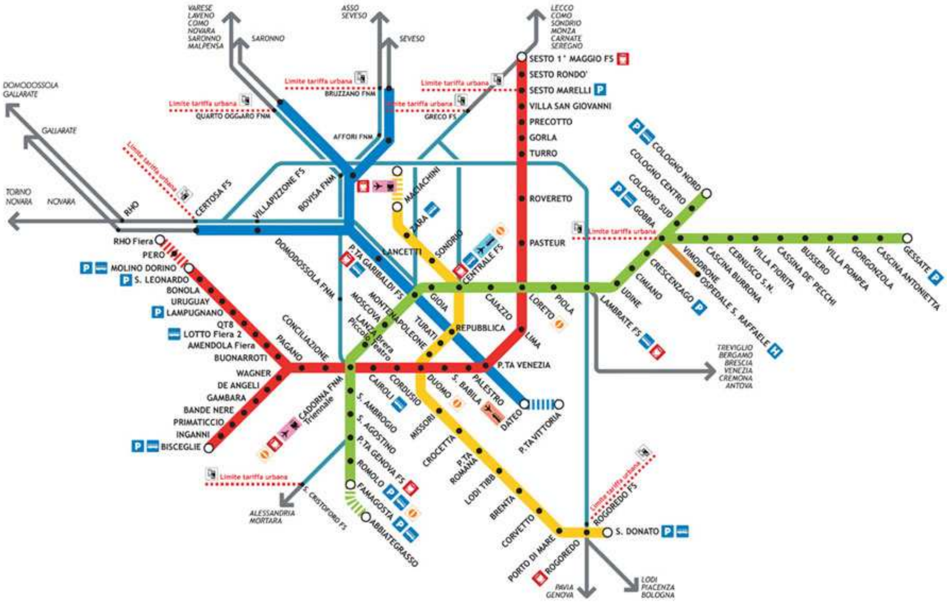
B A Word on the Coq Development 187
B.1 Overview of the Development 188
B.2 Basic Objects 188
B.2.1 Basic Types 188
B.2.2 Events and Program Order 190
B.2.3 Execution Witnesses 192
B.3 Architectures and Weak Memory Models 192
B.4 Proofs 194

Acknowledgements

Oh! I get by with a little help
from my friends,
Hmm, I get high with a little
help from my friends,
Hmm, I'm gonna try with a
little help from my friends...

*The Beatles—With a Little Help
from My Friends* [LM67c]

Ça vous occupera pendant que je fais le zouave au tableau! Et aussi, je ferai de bon cœur une dédicace manuscrite, ou je payerai une bière, à quiconque se sentirait floué de n'avoir pas été remercié ici.



Quand vous entendez ce petit tintement, tournez la page.

Pierre Clairambault · Jean-Baptiste Tristan · Delphine
Longuet · Derek Williams · Xavier Leroy · Samuel Mimram
· Fabrice Lemoine · Thomas Braibant · Paul-André Mellies

· Ahmed Bouajjani
· Amélie Mouton ·
Damien Doligez ·
Peter O'Hearn ·
Vincent Jacques ·
Boris Yakobowski
· Emmanuel Beffara ·
Florian Horn · Stéphane Zimmerman ·
Celtique · David Durrleman · Moscova ·
Marion Kermann · Gunther

Sikler ·
Proval · Contraintes ·
Julien Martin · Luc Maranget · Philippe
Paul · David Baelde · Daniel
Hirschkoff · Alexandre
Buisse · Yann
Strozecki · Aquinas
Hobor · Nadia Mesrar · Julien Cristau · Peter Sewell ·
Brice Goglin · Nicolas Kornman · Sylvain Soliman ·
Arnaud Sangnier · Yves-Alexis Perez · Assia Mahboubi ·

Samuel Hym
· Gérard Boudol · Florent
Bouchy · Vincent Siles ·
Nicolas Guenot · Claire
David · Sam Hocevar · Emmanuel
Jeandel · Dider Rémy · Pierre
Weis · Cédric Augonnet · Sandrine Blazy · Miles Alglave
· Maurice Herlihy · Juliette Simonet · Johan Marcellan ·

Christian Gatore · Mexico · Runtime · Arthur Loiret ·
Étienne Lozes · Zaynah Dargaye · Roberto Di Cosmo ·
Thomas Lepoutre · Gallium · Pierre Habouzit · Cristiano
Calcagno ·
Susmit Sarkar ·
Cyril Brulebois ·
Jules Villard ·
Vladimir
Katchadourian ·
Jérémie Leymarie
· Jean-Jacques
Lévy · Samuel Thibault · Sylvain Schmitz · Sylvie Burini

Part I

Preamble

Chapter 1

Introduction

Roll up! Roll up for the magical mystery tour, step right this way!

The Beatles—Magical Mystery Tour [LM67a]

1.1 Context

We give here a brief overview of the concepts that we will use throughout the manuscript. We illustrate these concepts with related work that we enjoyed reading, and that helped our comprehension of the vast area of weak memory models. This is nowhere near an exhaustive state of the art, but rather an overview of the work we found inspiring and from which our work inherits crucial ideas. We give in Fig. 1.1 a timeline of some of the related work we mention troughout the manuscript.

I guess that I had no idea how a program *should* behave. I may have been quite the only one; indeed programmers seem to expect certain behaviours and find others surprising when they run a program.

1.1.1 Early Days (1979 – 1995)

1.1.1.1 Sequential Consistency

Most of the time indeed, when writing a concurrent program, one expects (or would like) it to behave according to L. Lamport's *Sequential Consistency* (SC) [Lam79], where:

> *[. . .] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

2010	S. Adve and H.-J. Boehm's position paper [AB] S. Burckhardt et al's study of program transformations [BMS]
2009	PPC 2.06 [pow09] Intel rev. 30 [int09]
2008	S. Adve and H.-J. Boehm's C++ foundation [BA] S. Burckhardt and M. Musuvathi's Efficient Enumeration Techniques for TSO [BMa]
2007	PPC 2.05 [ppc07] Intel White Paper [int07] H.-J. Boehm's Pthread's Locks study [Boea] S. Burckhardt et al's Checkfence [BAM]
2006	Arvind and J.-W. Maessen's Instruction Reordering + Store Atomicity [AM]
2005	H.-J. Boehm's Threads Cannot Be Implemented As A Library [Boeb]
2004	S. Hangal et al's TSOTool [HVM$^+$]
2003	A. Adir et al's pre-cumulativity, view order style Power model [AAS03]
2002	Alpha's Fourth Edition [alp02]
2000	
1995	S. Adve and K. Gharachorloo's tutorial [AG95] K. Gharachorloo's thesis [Gha95]
1994	Sparc V9 [spa94b]
1993	S. Adve's thesis (DRF) [Adv93]
1992	W. Collier's framework[Col92] Sparc V8 [spa92]
1991	M. Herlihy's Wait-Free Synchronisation [Her]
1990	S. Adve and M. Hill's Weak Ordering [AH] M. Dubois and C. Scheurich's Memory Access Dependencies In Shared-Memory Multiprocessors [DS90]
1988	D. Shasha and M. Snir's delay set algorithm [SS]
1987	M. Dubois and C. Scheurich's Correct Memory Operation Of Cache Based Multiprocessors [SD87]
1985	
1979	L. Lamport's SC[Lam79]

Figure 1.1: Timeline of Selected Related Work

Consider for example the program given in Fig. 1.2(a), written in pseudo code. On P_0, we start with a store of value 1 to the memory location x, labelled (a), followed in program order by a load from memory location y into register `r1`, labelled (b). On P_1, we have a store of value 1 in memory location y, labelled (c), followed in program order by a load from memory location x into register `r2`, labelled (d). The registers are private to a processor, while the memory locations are shared.

We wonder whether the specified outcome—where `r1` on P_0 and `r2` on P_1 hold 0 in the end—can be observed if we assume SC as the execution model. The answer is no: SC authorises indeed only three final outcomes, depicted in Fig. 1.2(b). Suppose for example that the instructions on P_0 are executed before the ones on P_1; this corresponds to the first final state given in Fig. 1.2(b). In this case, the location x holds 1 because of the store (a) on P_0; the load (b) reads the initial value of y, which is 0: hence `r1` holds 0 in the end. Afterwards, P_1 executes its instructions. The store (c) writes 1 into y, and the load (d) reads from the last store to x, which is (a). Since (a) wrote 1 into x, `r2` holds 1 in the end.

1.1.1.2 Weak Memory Models

However, for matters of performance, modern processors may provide features that induce behaviours a machine with a SC model would never exhibit, as L. Lamport already exposed in [Lam79]:

> *For some applications, achieving sequential consistency may not be worth the price of slowing down the processors. In this case, one must be aware that conventional methods for designing multiprocess algorithms cannot be relied upon to produce correctly executing programs.*

Consider for example the test given in Fig. 1.2(c). It is the same test as in Fig. 1.2(a), but here it is written in x86 assembly. On P_0, the `MOV [x], $1` instruction corresponds to the store (a) of Fig. 1.2(a), and the `MOV EAX, [y]` corresponds to the load (b). Similarly on P_1, the `MOV [y], $1` instruction corresponds to the store (c), and the `MOV EAX, [x]` corresponds to the load (d). Although we expected only three possible outcomes when running this test, an x86 machine may exhibit the one which was specified in Fig. 1.2(a), because the store-load pairs on each processor may be reordered. Therefore, the load (b) on P_0 may occur before the store (c) on P_1: in that case, the load (b) reads the initial value of y, which is 0. Similarly, the load (c) on P_1 may occur before the store (a) on P_0, in which case (c) reads the initial value of x, which is 0. Thus, we obtain `r1=0` and `r2=0` as the final state.

Hence we cannot assume SC as the execution model of an x86 machine. A program running on a multiprocessor behaves *w.r.t.* the *memory model*

Init: `x=0; y=0;`

P_0	P_1
(a) x ← 1	(c) y ← 1
(b) r1 ← y	(d) r2 ← x

Observed? `r1=0; r2=0;`

(a) A Program

(a) (b) (c) (d): $r1 = 0 \wedge r2 = 1$
(c) (d) (a) (b): $r1 = 1 \wedge r2 = 0$
(a) (c) (b) (d): $r1 = 1 \wedge r2 = 1$

(b) The Three SC Outcomes for this program

```
{0:EAX=0; 1:EAX=0;
 x=0; y=0;}

 P0           | P1           ;
 MOV [x],$1   | MOV [y],$1   ;
 MOV EAX,[y]  | MOV EAX,[x]  ;

exists (0:EAX=0 /\ 1:EAX=0)
Condition is validated
```

(c) This Program in x86 Assembly

Figure 1.2: An Example

of the architecture. The memory models we studied are said to be *weak*, or *relaxed w.r.t.* to SC, because they allow more behaviours than SC. For example, such models may allow *instruction reordering* [AG95, AM]: reads and writes may not be preserved in the program order, as we just saw with the example of Fig. 1.2(a). Some of them [ita02, pow09] also relax the *store atomicity* [AG95, AM] constraint. A write may not be available to all processors at once: it could be *e.g.* at first initiated by a given processor, then commited to a store buffer or a cache, and finally *globally performed* to memory [DS90], at which point only it will be available for all processors to see. Hence the value of a given write may be available to certain processors sooner than to others.

Therefore one needs to understand precisely the definition and consequences of a given memory model in order to predict the possible outcomes of a running program. But some public documentations [int07, pow09] lack formal definitions of these models. The effort of writing correct concurrent programs is increased by the absence of precise, if not formal, definitions.

1.1.1.3 Synchronisation

SC is accepted as the most intuitive and simple memory model, as expressed *e.g.* by S. Adve and M. Hill in [AH]:

> *[...] programmers prefer to reason about sequentially consistent memory, rather than having to think about weaker memory, or even write buffers.*

Hence, for easier reasoning, the key idea is to specify a ”contract between software and hardware”, such that [AH]:

> *[...] software agrees to some formally specified constraints, and hardware agrees to appear sequentially consistent to at least the software that obeys those constraints.*

Data Race Freeness Guarantee As an example of such a contract, S. Adve and M. Hill proposed the Data Race Freeness Guarantee (DRF guarantee) [AH], ensuring a SC behaviour to a certain class of programs, the *correctly synchronised* [MPA] ones. They observe that [AH]:

> *The problem of maintaining sequential consistency manifests itself when two or more processors interact through memory operations on common variables.*

For example, the test of Fig. 1.2(a) illustrates exactly the above statement: indeed it may exhibit a non-SC outcome, and involves two processors communicating *via* two shared memory locations x and y.

So as to ensure that a program has a SC behaviour, S. Adve and M. Hill proposed the DRF$_0$ model. The DRF guarantee states that if there is a mean to *arbitrate* the *data races* in a program, then this program can only have SC executions. We consider that two memory accesses from distinct processors form a data race when they both are relative to the same location and one of them at least is a write.

In the example of Fig. 1.2(a), the store (a) on P_0 and the load (d) on P_1 form a data race. They belong to distinct processors, are both relative to the memory location x, and (a) is a write. Symmetrically, the store (c) on P_1 and the load (b) on P_0 form a data race relative to the memory location y.

Lock-Based Synchronisation The most common way of using the DRF guarantee is *mutual exclusion locks*. Locks ensure a certain execution order on the accesses they protect, hence can be used to arbitrate conflicts [HS08]. Locks authorise access to a given protected memory location if and only if a certain flag (the lock) is free; otherwise, the processor willing to access the location has to wait for the lock to be released by the processor currently holding it.

Consider for example the test of Fig. 1.2(a), where the accesses (a) and (d)—relative to x—are protected by the same lock ℓ, and the accesses (c) and (b)—relative to y—by another lock ℓ'. This means that when the lock ℓ is taken so that the access (a) is protected by ℓ, the access (d) has to wait for the lock ℓ to be free, and then to take it, to occur. The same reasoning applies for the accesses (c) and (b). The DRF guarantee ensures that the only possible outcomes of the modified program are the SC ones, given in Fig. 1.2(b).

However, locks suffer from several efficiency problems: they can induce some long delay of waiting for the lock to be free; see for example [HS08, 18.1.1, p.417].

Lock-Free Synchronisation In order to avoid the cost of locks, other synchronisation protocols were defined, such as *lock-free* synchronisation protocols. These protocols use synchronisation instructions provided by the architecture, such as *barriers* and composite synchronisation idioms, such as *test-and-set* or *compare-and-swap* [HS08, Sec. 5.6, p.112].

Consider for example the test of Fig. 1.2(a). Suppose that we place a barrier between (a) and (b) on P_0, and another one between (c) and (d) on P_1, such that these barriers prevent the reordering of write-read pairs. In this case, (a) and (b) cannot be reordered anymore; neither can (c) and (d). Therefore, the only possible outcomes of the modified program are the SC ones given in Fig. 1.2(b).

Contrarily to the DRF guarantee, lock-free protocols do not arbitrate data races. Thus they may improve the performance of the program. However, such a synchronisation requires a precise knowledge of the underlying memory model, since it uses barriers to prevent some weaknesses exhibited by the architecture. Thus the design and correctness of such protocols may be hard to prove, since the semantics of barriers is unclear, and the lack of formal definitions of weak memory models worsens the effort. The *delay set algorithm* by D. Shasha and M. Snir [SS] is an example of an early study of barriers placement, which assumes SC as the execution model.

1.1.2 Recent Days (2002 – 2010)

Most of the concurrent verification work suppose SC as memory model, probably because multiprocessors were not mainstream until recently. Nowadays however, since multiprocessors are widely spread, there is a recrudescent interest in such issues. Indeed, as exposed by S. Adve and H.-J. Boehm in [AB]:

> *The problematic transformations (e.g. reordering accesses to unrelated variables [. . .]) never change the meaning of single-threaded programs, but do affect multithreaded programs [. . .].*

1.1.2.1 The Problem Of Modelling

Most of the existing formalised yet intelligible memory models are *global time models*, for example Alpha's [alp02] and Sparc's [spa94a]. In such models, the actions or events issued by the different threads or processors can be embedded in a single order, representing the timeline in which these events occurred *w.r.t.* the whole system. By this we mean that all the participants involved agree on that global ordering of the events. Hence I believe that such models are the easiest to understand and to reason with, which may explain their prominence.

Indeed weak memory models are a challenging area of modelling. Not only because they are intricated models, but also because most of the vendors do not provide an abstract model to reason about the architecture their machines exhibit, apart from a few rare exceptions [alp02, spa94a]. Moreover, as S. Adve and H.-J. Boehm state [AB]:

> *Part of the challenge for hardware architects was the lack of clear memory models at the programming language level—it was unclear what programmers expected hardware to do.*

Thus modelling often resorts to intensive testing to support the theory. For example W. Collier's early work adopts a *black box testing* approach [Col92]. But this kind of approach is necessarily limited, and somewhat resembles wild guess, if it is not guided by a precise knowledge of either the architecture or the implementation.

1.1.2.2 Program Verification

The interest in the area of weak memory models is also probably renewed by recent research [Boeb, Boea, BPa] that takes the intricacies of weak memory models into account when proposing a new approach to software verification. In particular, S. Adve and H.-J. Boehm lead a group dedicated to these issues: they propose to enforce the DRF guarantee as a *sine qua non* condition, so as to make the verification of concurrent programs easier. S. Burckhardt *et al.* recently proposed a study of weak memory models in terms of rewriting rules [BMS]: thus the weak memory model is itself a program transformation. This elegant approach allows to reason about the correctness of several program transformations, *e.g.* induced by compilers.

1.2 Contribution

In that context, we tried to understand and formalise existing weak memory models. In order to do so, we first relied on the existing documentations. From this reading, we provided a generic framework which we showed to embrace several existing models. We then resorted to intensive testing to support our theory. To ease our pain while testing, we wrote a systematic testing tool, which computes small tests exercising one weakness of the architecture at a time. This allowed us to design a model for the Power architecture, which is an instance of our framework as well. Finally, we studied synchronisation in the context of weak memory models, whether lock-based or lock-free, from a generic point of view.

We briefly summarise our contribution here. We thus give an outline of the document, and some reading notes.

1.2.1 A Generic Framework

In Chap. 3, we present the generic axiomatic framework we developped in Coq [BC]. This is a global time model, widely inspired from Alpha and Sparc's documentation [alp02, spa94a]. We illustrate in Chap. 4 how to instantiate this framework to recover existing models, such as Sequential Consistency [Lam79], the Sparc hierarchy (*i.e.* Sun TSO, PSO and RMO) [spa94a], and Alpha [alp02]. We present some related work in Chap. 5, and detail some proofs in App. A.

Chap. 3 is the heart of this thesis, on which all our formal results are built. App. A can easily be skipped: the results which proofs are detailed in this chapter are described in Chap. 3 anyway.

1.2.2 A Testing Tool

We present in Chap. 7 our diy testing tool: diy computes small tests in Power or x86 assembly code and run them against hardware. The tests are computed from specifications proceeding from our generic model, presented in Chap. 3. We present in Sec. 8.1 the way we used our tool to study the Power architecture. We detail our experimental protocol, the characteristics of the machines we tested and the experimental results. Finally in Chap. 8, we provide and comment on the model for the Power architecture we deduced from our experiments, and we present some related work in Chap. 9.

Chap. 8 is an interesting instance of our framework. The description of the tool (Chap. 7) and the detailed experimental protocol (Sec. 8.1) may be skipped as the description of the Power model is self-contained. However, the justification for this model lies in those two chapters, because this is where we explain how we build our tests, and why they are relevant.

1.2.3 Synchronisation

We present in Chap. 11 a unifying approach to study both lock-based and lock-free synchronisations, which we illustrate by a proof that a generalised DRF guarantee holds for each instance of our generic model. We also provide a novel dual result on barrier placement. As both these results may enforce more synchronisation than necessary, we refine them in the style of D. Shasha and M. Snir [SS], by using *critical cycles*.

In addition, we present in Chap. 10 semantics for Power's locks and read-modify-write primitives. We show that these locks provide the refined generalised DRF guarantee. We show that the mapping of certain reads to read-modify-write primitives, coupled with non-cumulative barriers, restores Sequential Consistency, which spares the cost of cumulativity. Finally, we present some related work in Chap. 12.

1.2.4 Remainder

Chap. 2 presents some preliminary definitions and fundamental results on relations and orders, which we use throughout the document. This chapter may be skipped.

We give a conclusion and some perspectives of future work in Chap. 13 and Chap. 14 respectively. We say a word on our Coq development in App. B, which may be skipped.

Chapter 2

Preliminaries

Our basic objects are *relations* over some elements. We give in this chapter the fundamental definitions that we used. We also enunciate and prove the key lemmas that we used.

2.1 Relations

2.1.1 Basic Definitions

We use *homogeneous binary relations*, that is we use relations $\xrightarrow{r}$ such that r is, for a given set A, a set of pairs in $A \times A$.

We define the *domain* (written $\mathsf{domain}(\xrightarrow{r})$) and the *range* ($\mathsf{range}(\xrightarrow{r})$) of a relation $\xrightarrow{r}$ as follows :

Definition 1 (Domain and Range of a Relation)

$$\begin{aligned} \mathsf{domain}(\xrightarrow{r}) &\triangleq \{x \mid \exists y, x \xrightarrow{r} y\} \\ \mathsf{range}(\xrightarrow{r}) &\triangleq \{y \mid \exists x, x \xrightarrow{r} y\} \end{aligned}$$

We consider a relation $\xrightarrow{r}$ to be *transitive* (written $\mathsf{transitive}(\xrightarrow{r})$) when:

Definition 2 (Transitivity of a Relation)

$$\mathsf{transitive}(\xrightarrow{r}) \triangleq \forall xyz, (x \xrightarrow{r} y \wedge y \xrightarrow{r} z) \Rightarrow x \xrightarrow{r} z$$

We write $(\xrightarrow{r})^+$ for the *transitive closure* of $\xrightarrow{r}$, that is:

Definition 3 (Transitive Closure of a Relation)

$$x\,(\xrightarrow{r})^+\,y \quad\triangleq\quad x \xrightarrow{r} y \vee (\exists z, x\,(\xrightarrow{r})^+\,z \wedge z\,(\xrightarrow{r})^+\,y)$$

We consider a relation $\xrightarrow{r}$ to be *irreflexive* (written $\mathsf{irreflexive}(\xrightarrow{r})$) when:

Definition 4 (Irreflexivity of a Relation)

$$\mathsf{irreflexive}(\xrightarrow{\mathrm{r}}) \;\triangleq\; \neg(\exists x, x \xrightarrow{\mathrm{r}} x)$$

We consider a relation $\xrightarrow{\mathrm{r}}$ to be *total* over a set $\mathbb{E}$ (written $\mathsf{total}(\xrightarrow{\mathrm{r}}, \mathbb{E})$) when:

Definition 5 (Totality of a Relation)

$$\mathsf{total}(\xrightarrow{\mathrm{r}}, \mathbb{E}) \;\triangleq\; \forall (x, y) \in \mathbb{E} \times \mathbb{E}, x \xrightarrow{\mathrm{r}} y \vee y \xrightarrow{\mathrm{r}} x$$

We consider a relation $\xrightarrow{\mathrm{r}}$ to be *acyclic* (written $\mathsf{acyclic}(\xrightarrow{\mathrm{r}})$) when its transitive closure is irreflexive, *i.e.*:

Definition 6 (Acyclicity of a Relation)

$$\mathsf{acyclic}(\xrightarrow{\mathrm{r}}) \;\triangleq\; \neg(\exists x, x \,(\xrightarrow{\mathrm{r}})^{+}\, x)$$

2.1.2 Orders

We consider a relation $\xrightarrow{\mathrm{r}}$ to be a *partial order* over a set $\mathbb{E}$ when:

- the domain and the range of $\xrightarrow{\mathrm{r}}$ are included in $\mathbb{E}$,
- $\xrightarrow{\mathrm{r}}$ is transitive, and
- $\xrightarrow{\mathrm{r}}$ is irreflexive.

Formally, we have:

Definition 7 (Partial Order)

$$\begin{aligned} \mathsf{partial\text{-}order}(\xrightarrow{\mathrm{r}}, \mathbb{E}) \;\triangleq\;& (\mathsf{domain}(\xrightarrow{\mathrm{r}}) \cup \mathsf{range}(\xrightarrow{\mathrm{r}})) \subseteq \mathbb{E} \,\wedge \\ & \mathsf{transitive}(\xrightarrow{\mathrm{r}}) \wedge \mathsf{irreflexive}(\xrightarrow{\mathrm{r}}) \end{aligned}$$

We consider a relation $\xrightarrow{\mathrm{r}}$ to be a *total order* (or *linear strict order*) over $\mathbb{E}$ when:

- $\xrightarrow{\mathrm{r}}$ is a partial order over $\mathbb{E}$, and
- $\xrightarrow{\mathrm{r}}$ is total over $\mathbb{E}$.

Formally, we have:

Definition 8 (Total Order)

$$\mathsf{total\text{-}order}(\xrightarrow{\mathrm{r}}, \mathbb{E}) \;\triangleq\; \mathsf{partial\text{-}order}(\xrightarrow{\mathrm{r}}, \mathbb{E}) \wedge \mathsf{total}(\xrightarrow{\mathrm{r}}, \mathbb{E})$$

2.2 Linear Extension

We admit the following result: any partial order $\xrightarrow{r}$ can be *extended* into a total order $\xrightarrow{o}$. This total order $\xrightarrow{o}$ is a *linear extension* of $\xrightarrow{r}$. We write $\mathsf{linear\text{-}extension}(\xrightarrow{o}, \xrightarrow{r})$ to indicate that $\xrightarrow{o}$ is a linear extension of $\xrightarrow{r}$. Formally, we have:

Axiom 1 (Existence of a Linear Extension)

$$\begin{gathered}\forall \xrightarrow{r} \mathbb{E}, \mathsf{partial\text{-}order}(\xrightarrow{r}, \mathbb{E}) \Rightarrow \\ \exists \xrightarrow{o}, \mathsf{linear\text{-}extension}(\xrightarrow{o}, \xrightarrow{r}) \wedge \mathsf{total\text{-}order}(\xrightarrow{o}, \mathbb{E})\end{gathered}$$

We admit that a relation $\xrightarrow{r}$ is included in any of its linear extension:

Axiom 2 (Inclusion in its Linear Extensions)

$$\forall \xrightarrow{r}\xrightarrow{o}, \mathsf{linear\text{-}extension}(\xrightarrow{o}, \xrightarrow{r}) \Rightarrow (\xrightarrow{r} \subseteq \xrightarrow{o})$$

Finally, when $\xrightarrow{r}$ is already a total order, $\xrightarrow{r}$ has only one linear extension, which is itself:

Axiom 3 (Unicity of the Linear Extension of a Total Order)

$$\forall \xrightarrow{r} \mathbb{E}, \mathsf{total\text{-}order}(\xrightarrow{r}, \mathbb{E}) \Rightarrow (\forall \xrightarrow{o}, \mathsf{linear\text{-}extension}(\xrightarrow{o}, \xrightarrow{r}) \Rightarrow \xrightarrow{o} = \xrightarrow{r})$$

2.3 A Key Lemma

We define the sequence of two relations $\xrightarrow{r_1}$ and $\xrightarrow{r_2}$ (written $(\xrightarrow{r_1}; \xrightarrow{r_2})$) as follows:

Definition 9 (Sequence of Relations)

$$x(\xrightarrow{r_1}; \xrightarrow{r_2})y \triangleq \exists z, x \xrightarrow{r_1} z \wedge z \xrightarrow{r_2} y$$

We will often show that the acyclicity of a certain relation $\xrightarrow{r_1}$ implies the acyclicity of another one $\xrightarrow{r_2}$. In order to do so, we reason by contradiction: we suppose that $\xrightarrow{r_2}$ has a cycle, and show that in this case, $\xrightarrow{r_1}$ has a cycle as well, which contradicts its acyclicity.

In order to prove this, we use this key lemma, which states that if there is a cycle in the union of two irreflexive relations, there is a cycle in their sequence as well:

Lemma 1 (Cycle in Union Implies Cycle in Sequence)

$$\begin{gathered}\forall \xrightarrow{r_1}\xrightarrow{r_2}, \mathsf{irreflexive}(\xrightarrow{r_1}) \wedge \mathsf{irreflexive}(\xrightarrow{r_2}) \wedge \\ \mathsf{transitive}(\xrightarrow{r_1}) \wedge \mathsf{transitive}(\xrightarrow{r_2}) \wedge \\ \neg(\mathsf{acyclic}(\xrightarrow{r_1} \cup \xrightarrow{r_2})) \Rightarrow \neg(\mathsf{acyclic}(\xrightarrow{r_1}; \xrightarrow{r_2}))\end{gathered}$$

We need to establish a few more lemmas to prove this result.

2.3.1 Hexa Relation

We define the reflexive closure over relations as follows:

Definition 10 (Reflexive Closure)

$$x\,(\xrightarrow{\text{r}})^?\,y \;\triangleq\; x \xrightarrow{\text{r}} y \vee x = y$$

This means that two elements x and y are in $(\xrightarrow{\text{r}})^?$ when they are in $\xrightarrow{\text{r}}$ or $x = y$. Note that for any relation $\xrightarrow{\text{r}}$ and for all x, we have $x\,(\xrightarrow{\text{r}})^?\,x$.

We define the binary operator phx over relations as follows:

Definition 11 (Pre Hexa)

$$\mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2}) \;\triangleq\; ((\xrightarrow{\text{r}_1}; \xrightarrow{\text{r}_2})^+)^?$$

Note that, since phx is a reflexive closure, we have $(x, x) \in \mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})$ for all $\xrightarrow{\text{r}_1}$, $\xrightarrow{\text{r}_2}$ and x. Moreover, a phx construction is trivially transitive.

We define the binary operator hx over relations as follows:

Definition 12 (Hexa)

$$\mathsf{hx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2}) \;\triangleq\; (\xrightarrow{\text{r}_2})^?; \mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2}); (\xrightarrow{\text{r}_1})^?$$

We show that the hx of two transitive relations is a transitive relation:

Lemma 2 (Hexa Is Transitive)

$$\forall \xrightarrow{\text{r}_1}\xrightarrow{\text{r}_2}, \mathsf{transitive}(\xrightarrow{\text{r}_1}) \wedge \mathsf{transitive}(\xrightarrow{\text{r}_2}) \Rightarrow \mathsf{transitive}(\mathsf{hx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2}))$$

Proof *Suppose x, y and z such that $(x, y) \in \mathsf{hx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})$ and $(y, z) \in \mathsf{hx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})$. There exist z_1, z_2, z_3 and z_4 such that*

$$x\,(\xrightarrow{\text{r}_2})^?\,z_1\,\mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})z_2\,(\xrightarrow{\text{r}_1})^?\,y\,(\xrightarrow{\text{r}_2})^?\,z_3\,\mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})z_4\,(\xrightarrow{\text{r}_1})^?\,z$$

We have $z_2\,(\xrightarrow{\text{r}_1})^?\,y\,(\xrightarrow{\text{r}_2})^?\,z_3$, hence $(z_2, z_3) \in \mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})$. Since a phx construction is transitive, we have $(z_1, z_4) \in \mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})$, hence $x\,(\xrightarrow{\text{r}_2})^?\,z_1\,\mathsf{phx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})z_4\,(\xrightarrow{\text{r}_1})^?\,z$. Therefore, by the definition of hx, we have $(x, z) \in \mathsf{hx}(\xrightarrow{\text{r}_1}, \xrightarrow{\text{r}_2})$.

We show that when two relations $\xrightarrow{\text{r}_1}$ and $\xrightarrow{\text{r}_2}$ are both transitive, a path in their union is a path in $\mathsf{hx}(\xrightarrow{\text{r}_2}, \xrightarrow{\text{r}_1})$:

Lemma 3 (Path in Union Implies Path in Hexa)

$$\forall \xrightarrow{\text{r}_1}\xrightarrow{\text{r}_2}, \mathsf{transitive}(\xrightarrow{\text{r}_1}) \wedge \mathsf{transitive}(\xrightarrow{\text{r}_2}) \Rightarrow ((\xrightarrow{\text{r}_1} \cup \xrightarrow{\text{r}_2})^+ \subseteq \mathsf{hx}(\xrightarrow{\text{r}_2}, \xrightarrow{\text{r}_1}))$$

Proof *Suppose x and y such that $x\,(\xrightarrow{\text{r}_1} \cup \xrightarrow{\text{r}_2})^+\,y$. Let us reason by induction over this statement.*

- *Suppose $x(\xrightarrow{\text{r}_1} \cup \xrightarrow{\text{r}_2})y$.*

  - *Suppose $x \xrightarrow{\mathrm{r}_1} y$. We want to show that $(x,y) \in \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1})$, i.e. there exist z_1 and z_2 such that $x\,(\xrightarrow{\mathrm{r}_1})^?\, z_1\, \mathsf{phx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}) z_2\, (\xrightarrow{\mathrm{r}_2})^?\, y$. We know that $x\,(\xrightarrow{\mathrm{r}_1})^?\, y$ since $x \xrightarrow{\mathrm{r}_1} y$. Hence we can take $z_1 = y$. Since $(y,y) \in \mathsf{phx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1})$, we can take $z_2 = y$. Indeed we have $y\,(\xrightarrow{\mathrm{r}_2})^?\, y$.*

  - *Suppose $x \xrightarrow{\mathrm{r}_2} y$, we can take $z_1 = z_2 = x$.*

- *In the inductive case, we have z such that $x\, \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}) z$ and $z\, \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}) y$. By Lem. 2, we have the result.*

We show that if $(\xrightarrow{\mathrm{r}_2}; \xrightarrow{\mathrm{r}_1})$ is acyclic, then $(\mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}); \xrightarrow{\mathrm{r}_1})$ is irreflexive, provided $\xrightarrow{\mathrm{r}_1}$ is irreflexive and transitive:

Lemma 4 (Hexa Right)

$$\begin{gathered} \forall \xrightarrow{\mathrm{r}_1} \xrightarrow{\mathrm{r}_2} xy, \mathsf{irreflexive}(\xrightarrow{\mathrm{r}_1}) \wedge \mathsf{transitive}(\xrightarrow{\mathrm{r}_1}) \wedge \\ (x,y) \in \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}) \wedge y \xrightarrow{\mathrm{r}_1} x \Rightarrow \exists z, z\, (\xrightarrow{\mathrm{r}_2}; \xrightarrow{\mathrm{r}_1})^+\, z \end{gathered}$$

Proof *Suppose x and y such that $(x,y) \in \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1})$ and $y \xrightarrow{\mathrm{r}_1} x$. There are z_1 and z_2 such that $x\,(\xrightarrow{\mathrm{r}_1})^?\, z_1\, \mathsf{phx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}) z_2\, (\xrightarrow{\mathrm{r}_2})^?\, y$. The case when $x = y$ is a direct contradiction to the fact that $\xrightarrow{\mathrm{r}_1}$ is irreflexive, since $y \xrightarrow{\mathrm{r}_1} x$ by hypothesis. Otherwise, we have $z_1\, (\xrightarrow{\mathrm{r}_2}; \xrightarrow{\mathrm{r}_1})^+\, z_1$.*

We show that if $(\xrightarrow{\mathrm{r}_2}; \xrightarrow{\mathrm{r}_1})$ is acyclic, then $(\xrightarrow{\mathrm{r}_1}; \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}))$ is irreflexive, provided $\xrightarrow{\mathrm{r}_2}$ is irreflexive and transitive:

Lemma 5 (Hexa Left)

$$\begin{gathered} \forall \xrightarrow{\mathrm{r}_1} \xrightarrow{\mathrm{r}_2} xy, \mathsf{irreflexive}(\xrightarrow{\mathrm{r}_2}) \wedge \mathsf{transitive}(\xrightarrow{\mathrm{r}_2}) \wedge \\ x \xrightarrow{\mathrm{r}_2} y \wedge (y,x) \in \mathsf{hx}(\xrightarrow{\mathrm{r}_2}, \xrightarrow{\mathrm{r}_1}) \Rightarrow \exists z, z\, (\xrightarrow{\mathrm{r}_2}; \xrightarrow{\mathrm{r}_1})^+\, z \end{gathered}$$

Proof *The proof is similar to the proof of Lem. 4 above.*

2.3.2 Proof of the Result

We want to prove the Lem. 1:

$$\begin{gathered} \forall \xrightarrow{\mathrm{r}_1} \xrightarrow{\mathrm{r}_2}, \mathsf{irreflexive}(\xrightarrow{\mathrm{r}_1}) \wedge \mathsf{irreflexive}(\xrightarrow{\mathrm{r}_2}) \wedge \\ \mathsf{transitive}(\xrightarrow{\mathrm{r}_1}) \wedge \mathsf{transitive}(\xrightarrow{\mathrm{r}_2}) \wedge \\ \neg(\mathsf{acyclic}(\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2})) \Rightarrow \neg(\mathsf{acyclic}(\xrightarrow{\mathrm{r}_1}; \xrightarrow{\mathrm{r}_2})) \end{gathered}$$

Proof *Since $\neg(\mathsf{acyclic}(\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2}))$, there is x such that $x\, (\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2})^+\, x$. Therefore, there is z such that $x(\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2})z$ and $(z\, (\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2})^+\, x) \vee (x = z)$. We do a case disjunction over this last statement.*

- *Suppose $x = z$. As $x(\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2})z$, we have $x(\xrightarrow{\mathrm{r}_1} \cup \xrightarrow{\mathrm{r}_2})x$. But since $\xrightarrow{\mathrm{r}_1}$ and $\xrightarrow{\mathrm{r}_2}$ are both irreflexive, their union is irreflexive as well, hence the result.*

- *Suppose* $(z\,(\xrightarrow{r_1} \cup \xrightarrow{r_2})^+\, x)$. *In this case, we have* $(z\,((\xrightarrow{r_1})^+ \cup (\xrightarrow{r_2})^+)^+\, x)$. *By Lem. 3, we have* $(z, x) \in \mathsf{hx}((\xrightarrow{r_2})^+, (\xrightarrow{r_1})^+)$.

  Moreover, we know that $x(\xrightarrow{r_1} \cup \xrightarrow{r_2})z$. *Let us do a case disjunction over this statement.*

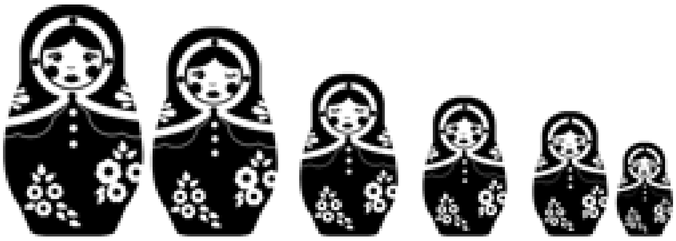
  - *Suppose* $x \xrightarrow{r_1} z$. *By Lem. 4 applied to* $(z, x) \in \mathsf{hx}((\xrightarrow{r_2})^+, (\xrightarrow{r_1})^+)$ *and* $x \xrightarrow{r_1} z$, *we have the result.*
  - *Suppose now* $x \xrightarrow{r_2} z$. *By Lem. 5 applied to* $x \xrightarrow{r_2} z$ *and* $(z, x) \in \mathsf{hx}((\xrightarrow{r_2})^+, (\xrightarrow{r_1})^+)$, *we have the result.*

Part II

A Generic Framework For Weak Memory Models

We present here a generic framework designed to reason on weak memory models. Though some public documentations, *e.g.* Intel [int07] and Power [pow09], lack formal definitions of these models, others—such as Alpha [alp02] and Sparc [spa94a]—provide a precise definition of the model their processors exhibit. Our generic framework is widely inspired of the common style of Alpha and Sparc's documentations, in that we use a *global time axiomatic* model. However, Alpha and Sparc consider the stores to be *atomic*. We adapted the style of their model to allow the store atomicity relaxation, as does *e.g.* Power. In addition, we took care to minimise the number and the complexity of our axioms, so that they are easier to understand.

We present in Chap. 3 the objects, terms and axioms of our framework. We illustrate in Chap. 4 how to instantiate its parameters to produce several well known models, namely *Sequential Consistency* [Lam79], the Sparc hierarchy (*i.e.* TSO, PSO and RMO) [spa94a], and Alpha [alp02]. We conclude with a presentation of some related work in Chap. 5.



Chapter 3

A Generic Framework

3.1 Basic Objects

A memory model determines whether a candidate execution of a program is *valid*. We consider an execution of a given program to be valid when the read and write memory events associated with the instructions of the program follow a single global timeline, *i.e.* can be embedded in a single partial order. This order represents the timeline in which these events are *globally performed*, which means that we embed them in the order when we reach the point in time where all processors involved have to take these events into account.

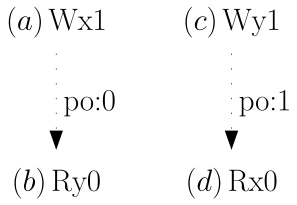
We illustrate our model with *litmus* tests, which are simple tests in pseudo- or assembly code. Fig. 3.1(a) shows such a test, with an initial state (which gathers the initial values of registers and memory locations used in the test), a program in pseudo- or assembly code, and a final condition on registers and memory (we write x, y for memory locations and `r1`, `r2` for registers). We give in Fig. 3.1(b) an execution associated with this test. The validity of this execution is relative to the architecture on which it runs: it is valid on an architecture such as x86, whereas it is invalid on SC.

Init: `x=0; y=0;`

P_0	P_1
(a) `x ← 1`	(c) `y ← 1`
(b) `r1 ← y`	(d) `r2 ← x`

Observed? `r1=0; r2=0;`

(a) A Program



(b) Events and Program Order

Figure 3.1: A Program and an Event Structure

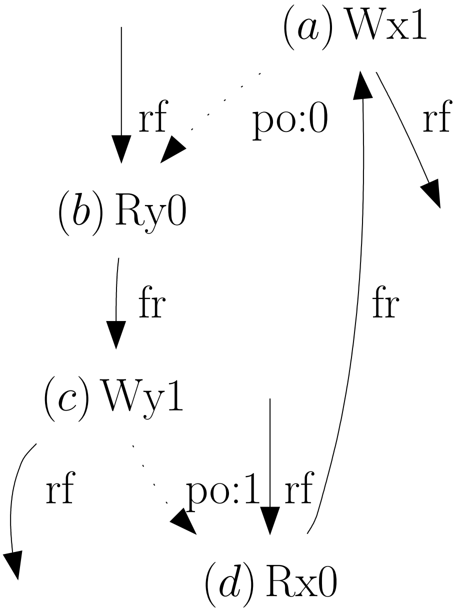


Figure 3.2: An Execution for the Program of Fig. 3.1

3.1.1 Events and Program Order

Rather than dealing directly with programs, our models are expressed in terms of the *events* $\mathbb{E}$ occurring in a candidate execution. A *memory event m* represents a memory access, specified by its direction (write or read), its location $\mathsf{loc}(m)$, its processor $\mathsf{proc}(m)$, and a unique label. For example, the store to x marked (a) in Fig. 3.1(a) generates the event (a) Wx in Fig. 3.1(b). Henceforth, we write r (resp. w) for a read (resp. write) event. We write $\mathbb{M}_\ell$ (resp. $\mathbb{R}_\ell$, $\mathbb{W}_\ell$) for the set of memory events (resp. reads, writes) to a location ℓ (we omit ℓ when quantifying over all of them). We give a table of notations for these sets of events, and the corresponding cartesian products in Fig. 3.4.

The models are defined in terms of binary relations over these events, and Fig. 3.3 shows a table of the relations we use.

The *program order* $\xrightarrow{\text{po}}$ is a total order amongst the events from the same processor that never relates events from different processors. It reflects the sequential execution of instructions on a single processor: given two instruction execution instances i_1 and i_2 that generate events e_1 and e_2, $e_1 \xrightarrow{\text{po}} e_2$ means that a sequential processor would execute i_1 before i_2. When instructions may perform several memory accesses, we take intra-instruction dependencies [SSZN$^+$] into account to build a more precise order.

Hence we describe a program by an *event structure*, which collects the memory events issued by the instructions of this program, and the program order relation, which lifts the program order between instructions to the events' level:

Definition 13 (Event Structure)

$$E \triangleq (\mathbb{E}, \xrightarrow{\text{po}})$$

Consider for example the test given in Fig. 3.1(a). We give in Fig. 3.1(b) an associated event structure. For example, to the store instruction marked (a) on P_0, we associate the write event (a) Wx in Fig. 3.1(b). To the load instruction marked (b) on P_0, we associate the read event (b) Ry in Fig. 3.1(b). Since these two instructions are in program order on P_0, the associated events are related by the $\xrightarrow{\text{po}}$ relation. The reasoning is similar on P_1.

3.2 Execution Witnesses

Although $\xrightarrow{\text{po}}$ conveys important features of program execution, *e.g.* branch resolution, it does not characterise an execution. Indeed, on a weak memory model, the events in program order may be reordered in an execution. Moreover, we need to describe the communication between distinct processors during the execution of a program. Hence, in order to describe an execution, we postulate two relations $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ over memory events.

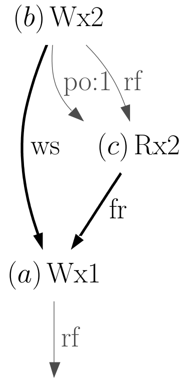
Name	Notation	Comment	Sec.
program order	$m_1 \xrightarrow{\text{po}} m_2$	per-processor total order	3.1.1
dependencies	$m_1 \xrightarrow{\text{dp}} m_2$	included in $\xrightarrow{\text{po}}$, source is a read	3.4.1.3
po-loc	$m_1 \xrightarrow{\text{po-loc}} m_2$	program order restricted to the same location, included in $\xrightarrow{\text{po}}$	3.4.1.1
preserved program order	$m_1 \xrightarrow{\text{ppo}} m_2$	pairs maintained in program order, included in $\xrightarrow{\text{po}}$	3.3.2
read-from map	$w \xrightarrow{\text{rf}} r$	links a write to a read reading its value	3.2.1
external read-from map	$w \xrightarrow{\text{rfe}} r$	$\xrightarrow{\text{rf}}$ between events from distinct processors	3.3.1
internal read-from map	$w \xrightarrow{\text{rfi}} r$	$\xrightarrow{\text{rf}}$ between events from the same processor	3.3.1
global read-from map	$w \xrightarrow{\text{grf}} r$	$\xrightarrow{\text{rf}}$ considered global	3.3.1
write serialisation	$w_1 \xrightarrow{\text{ws}} w_2$	total order on writes to the same location	3.2.2
from-read map	$r \xrightarrow{\text{fr}} w$	r reads from a write preceding w in $\xrightarrow{\text{ws}}$	3.2.3
barriers	$m_1 \xrightarrow{\text{ab}} m_2$	ordering induced by barriers	3.3.3
global happens-before	$m_1 \xrightarrow{\text{ghb}} m_2$	union of global relations	3.3.4
communication	$m_1 \xrightarrow{\text{com}} m_2$	shorthand for m_1 $(\xrightarrow{\text{rf}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}})$ m_2	3.2.4

Figure 3.3: Table of Relations

Name	Notation	Comment
memory events	$\mathbb{M}$	all memory events
memory events to the same location	$\mathbb{M}_\ell$	memory events relative to the location ℓ
read events, reads	$\mathbb{R}$	memory events that are reads
reads from the same location	$\mathbb{R}_\ell$	reads from the location ℓ
write events, writes	$\mathbb{W}$	memory events that are writes
writes to the same location	$\mathbb{W}_\ell$	writes to the location ℓ
memory pairs	$\mathbb{M} \times \mathbb{M}$	pairs of any memory events in program order
memory pairs to the same location	$\mathbb{M}_\ell \times \mathbb{M}_\ell$	pairs of any memory events to the same location in program order
read-read pairs	$\mathbb{R} \times \mathbb{R}$	pairs of reads in program order
read-read pairs to the same location	$\mathbb{R}_\ell \times \mathbb{R}_\ell$	pairs of reads from the same location in program order
read-write pairs	$\mathbb{R} \times \mathbb{W}$	read followed by write in program order
read-write pairs to the same location	$\mathbb{R}_\ell \times \mathbb{W}_\ell$	read followed by write to the same location in program order
write-write pairs	$\mathbb{W} \times \mathbb{W}$	pairs of writes in program order
write-write pairs to the same location	$\mathbb{W}_\ell \times \mathbb{W}_\ell$	pairs of writes to the same location in program order
write-read pairs	$\mathbb{W} \times \mathbb{R}$	write followed by read in program order
write-read pairs to the same location	$\mathbb{W}_\ell \times \mathbb{R}_\ell$	write followed by read from the same location in program order

Figure 3.4: Table of Notations for events and pairs of events

P_0	P_1
(a) `x` $\leftarrow$ `1`	(b) `x` $\leftarrow$ `2`
	(c) `r1` $\leftarrow$ `x`
Allowed: `r1=2; x=1`	



(a) A Program

(b) An Execution Witness

Figure 3.5: A Program and a Candidate Execution

3.2.1 Read-From Map

We write $w \xrightarrow{\text{rf}} r$ to mean that r loads the value stored by w (so w and r must share the same location). In any execution, given a read r there exists a unique write w such that $w \xrightarrow{\text{rf}} r$ (w can be an *init* store when r loads from the initial state). Thus, $\xrightarrow{\text{rf}}$ must be well formed following the $\mathsf{wf\text{-}rf}$ predicate:

Definition 14 (Well-Formed Read-From Map)

$$\mathsf{wf\text{-}rf}(\xrightarrow{\text{rf}}) \triangleq \left(\xrightarrow{\text{rf}} \subseteq \bigcup_{\ell}(\mathbb{W}_\ell \times \mathbb{R}_\ell)\right) \wedge (\forall r,\ \exists!\, w.\ w \xrightarrow{\text{rf}} r)$$

Consider the example given in Fig. 3.5(a). In the associated execution given in Fig. 3.5(b), the read (c) from x on P_1 reads its value from the write (b) to x on P_1. Hence we have a $\xrightarrow{\text{rf}}$ relation between them, depicted in the execution: $(b) \xrightarrow{\text{rf}} (c)$.

3.2.2 Write Serialisation

We assume all values written to a given location ℓ to be serialised, following a *coherence order*. This property is widely assumed by modern architectures. We define $\xrightarrow{\text{ws}}$ as the union of the coherence orders for all memory locations, which must be well formed following the $\mathsf{wf\text{-}ws}$ predicate:

Definition 15 (Well-Formed Write Serialisation)

$$\mathsf{wf\text{-}ws}(\xrightarrow{\text{ws}}) \triangleq \left(\xrightarrow{\text{ws}} \subseteq \bigcup_{\ell}(\mathbb{W}_\ell \times \mathbb{W}_\ell)\right) \wedge \left(\forall \ell.\ \mathsf{total\text{-}order}\left(\xrightarrow{\text{ws}}, (\mathbb{W}_\ell \times \mathbb{W}_\ell)\right)\right)$$

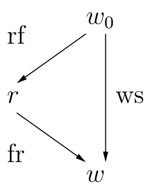


Figure 3.6: $\xrightarrow{\text{fr}}$ Proceeds From $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$

Consider the example given in Fig. 3.5(a). In the associated execution given in Fig. 3.5(b), the write (b) to x on P_1 hits the memory before the write (a) to x on P_0. Hence we have a $\xrightarrow{\text{ws}}$ relation between them, depicted in the execution: $(b) \xrightarrow{\text{ws}} (a)$.

As we shall see in Sec. 3.3.1, we will embed the write events in our global timeline according to the write serialisation.

3.2.3 From-Read Map

We define the derived relation $\xrightarrow{\text{fr}}$[ABJ$^+$] which gathers all pairs of reads r and writes w such that r reads from a write that is before w in $\xrightarrow{\text{ws}}$, as depicted in Fig. 3.6. Intuitively, a read r is in $\xrightarrow{\text{fr}}$ with a write w when r reads from a write that hit the memory before w did:

Definition 16 (From-Read Map)

$$r \xrightarrow{\text{fr}} w \ \triangleq\ \exists\, w'.\, w' \xrightarrow{\text{rf}} r \wedge w' \xrightarrow{\text{ws}} w$$

Consider the example given in Fig. 3.5(a). In the associated execution given in Fig. 3.5(b), the write (b) to x on P_1 hits the memory before the write (a) to x on P_0, *i.e.* $(b) \xrightarrow{\text{ws}} (a)$. Moreover, the read (c) from x on P_1 reads its value from the write (b) to x on P_1, *i.e.* $(b) \xrightarrow{\text{rf}} (c)$. Hence we have a $\xrightarrow{\text{fr}}$ relation between (c) and (a) (*i.e.* $(c) \xrightarrow{\text{fr}} (a)$) because (c) reads from a write which is older than (a) in the write serialisation.

As we shall see in Sec. 3.3.1, we will use the $\xrightarrow{\text{fr}}$ relation to include the read events in our global timeline.

3.2.4 All Together

Given a certain event structure E, we call the $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ relations the *communication* relations, and we write $\xrightarrow{\text{com}}$ for their union:

Definition 17 (Communication Relation)

$$\xrightarrow{\text{com}} \ \triangleq\ \xrightarrow{\text{rf}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}}$$

We define an *execution witness* X associated with an event structure E as follows:

Definition 18 (Execution Witness)

$$X \triangleq (\xrightarrow{\text{rf}}, \xrightarrow{\text{ws}})$$

For example, we give in Fig. 3.2 an execution witness associated with the program of Fig. 3.1(a). The set of events is $\{(a), (b), (c), (d)\}$, the program order is $(a) \xrightarrow{\text{po}} (b), (c) \xrightarrow{\text{po}} (d)$. Since the initial state is implicitly a write preceding any other write to the same location in the write serialisation, the only communication arrows we have between the events of this execution are $(b) \xrightarrow{\text{fr}} (c)$ and $(d) \xrightarrow{\text{fr}} (a)$.

The well-formedness predicate wf on execution witnesses is the conjunction of those for $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{rf}}$. We write $\mathsf{rf}(X)$ (resp. $\mathsf{ws}(X)$, $\mathsf{po}(X)$) to extract the $\xrightarrow{\text{rf}}$ (resp. $\xrightarrow{\text{ws}}$, $\xrightarrow{\text{po}}$) relation from a given execution witness X. When X is clear from the context, we may write $\xrightarrow{\text{rf}}$ instead of $\mathsf{rf}(X)$ for example.

Definition 19 (Well-Formed Execution Witness)

$$\mathsf{wf}(X) \triangleq \mathsf{wf\text{-}rf}(\mathsf{rf}(X)) \wedge \mathsf{wf\text{-}ws}(\mathsf{ws}(X))$$

3.3 Global Happens-Before

We consider an execution to be valid when we can embed the memory events of this execution is a single global timeline. By global we mean that the memory events are the events relative to memory actions, in a way that every processor involved has to take them into account. Therefore, we do not consider the events relative to store buffers or caches, but rather we wait until these events hit the main memory. Thus, we focus on the history of the system from the main memory's point of view.

Hence, an execution witness is valid if the memory events can be embedded in an acyclic *global happens-before* relation $\xrightarrow{\text{ghb}}$ (together with two auxiliary conditions detailed in Sec. 3.4). This order corresponds roughly to the vendor documentation concept of memory events being *globally performed* [pow09, DS90]: a write in $\xrightarrow{\text{ghb}}$ represents the point in global time when this write becomes visible to all processors; whereas a read in $\xrightarrow{\text{ghb}}$ represents the point in global time when the read takes place. We will formalise this notion later on, at Sec. 3.3.4.

In order to do so, we present first the choices as to which relations we include in $\xrightarrow{\text{ghb}}$ (*i.e.* which we consider to be in global time). Thereby we define a class of models. In the following, we will call a relation *global* when it is included in $\xrightarrow{\text{ghb}}$. Intuitively, a relation is considered global if the participants of the system have to take it into account to build a valid execution.

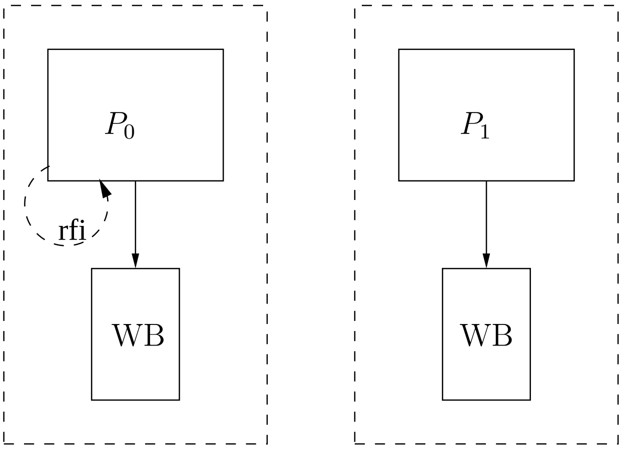


Figure 3.7: Store Buffering

3.3.1 Globality

Writes are not necessarily globally performed at once. Some architectures allow *store buffering* (or *read own writes early* [AG95]): the processor issuing a given write can read its value before any other participant has access to it. Other architectures allow two processors sharing a cache to read a write issued by their neighbour *w.r.t.* the cache hierarchy before any other participant that does not share the same cache (a case of *store atomicity relaxation*, or *read others' writes early* [AG95]).

In our class of models, $\xrightarrow{\text{ws}}$ is always included in $\xrightarrow{\text{ghb}}$. Indeed, the write serialisation for a given location ℓ is by definition the order in which writes to ℓ are globally performed.

Yet, $\xrightarrow{\text{rf}}$ is not necessarily included in $\xrightarrow{\text{ghb}}$. Let us distinguish between internal (resp. external) $\xrightarrow{\text{rf}}$, when the two events in $\xrightarrow{\text{rf}}$ are on the same (resp. distinct) processor(s), written $\xrightarrow{\text{rfi}}$ (resp. $\xrightarrow{\text{rfe}}$) :

Definition 20 (Internal and External Read-From Map)

$$\begin{aligned} w \xrightarrow{\text{rfi}} r &\triangleq w \xrightarrow{\text{rf}} r \wedge \mathsf{proc}(w) = \mathsf{proc}(r) \\ w \xrightarrow{\text{rfe}} r &\triangleq w \xrightarrow{\text{rf}} r \wedge \mathsf{proc}(w) \neq \mathsf{proc}(r) \end{aligned}$$

We model the store buffering by $\xrightarrow{\text{rfi}}$ being not included in $\xrightarrow{\text{ghb}}$, as depicted in Fig. 3.7. Indeed, the communication between a write from a given processor's store buffer and a read does not influence the execution of another processor, because the write has not hit the main memory yet. Therefore, this communication, modelled by $\xrightarrow{\text{rfi}}$ is private to the processor issuing the write, and we do not embed it in our global timeline.

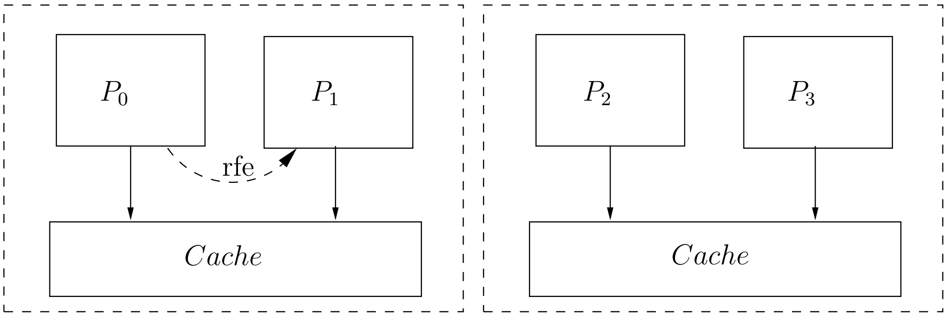


Figure 3.8: Store Atomicity Relaxation

Similarly, we model the store atomicity relaxation by $\xrightarrow{\text{rfe}}$ being not global, as depicted in Fig. 3.8. Indeed, the communication between two processors *via* a shared cache may not influence the execution of another processor, because the write has not hit the main memory yet. Therefore, this communication, modelled by $\xrightarrow{\text{rfe}}$ is private to the two communicating processors, and we do not embed it in our global timeline.

We write $\xrightarrow{\text{grf}}$ for the subrelation of $\xrightarrow{\text{rf}}$ included in $\xrightarrow{\text{ghb}}$.

In our framework, $\xrightarrow{\text{fr}}$ is always included in $\xrightarrow{\text{ghb}}$. Indeed, as $r \xrightarrow{\text{fr}} w$ means that the write w' from which r reads is globally performed before w, it forces the read r to be globally performed (since a read is globally performed as soon as it is performed) before w is globally performed.

3.3.2 Preserved Program Order

In any given architecture, certain pairs of events in the program order are guaranteed to occur in this order. We postulate a global relation $\xrightarrow{\text{ppo}}$ gathering all such pairs. For example, the execution witness in Fig. 3.2 is only valid if the writes and reads relative to different locations on each processor have been reordered. Indeed, if these pairs were forced to be in program order, we would have a cycle in $\xrightarrow{\text{ghb}}$: $(a) \xrightarrow{\text{ppo}} (b) \xrightarrow{\text{fr}} (c) \xrightarrow{\text{ppo}} (d) \xrightarrow{\text{fr}} (a)$. Such a cycle contradicts the validity of the execution, hence the execution depicted in Fig. 3.2 is not valid on an architecture such as SC, which maintains the write-read pairs in program order.

3.3.3 Barriers Constraints

Architectures also provide *barrier* instructions, *e.g.* the Power `sync`, to enforce ordering between pairs of events. We postulate a global relation $\xrightarrow{\text{ab}}$ gathering all such pairs.

A barrier is *non-cumulative* when it induces a relation $\xrightarrow{\text{r}}$ ordering certain pairs of events surrounding (in program order) the barrier; we write $\mathsf{NC}(\xrightarrow{\text{r}})$

in this case. The relation $\xrightarrow{\text{r}}$ induced by a barrier is *A-cumulative* $w.r.t.$ a certain subrelation $\xrightarrow{\text{s}}$ of $\xrightarrow{\text{rf}}$ (resp. *B-cumulative*), written $\mathsf{AC}(\xrightarrow{\text{r}}, \xrightarrow{\text{s}})$ (resp. $\mathsf{BC}(\xrightarrow{\text{r}}, \xrightarrow{\text{s}})$), when the barrier makes certain writes atomic ($e.g.$ by flushing the store buffers and caches).

Intuitively, a barrier b placed between two instructions i_1 and i_2 in program order has a non-cumulative ordering over the associated events m_1 and m_2 when it makes the program order between m_1 and m_2 visible to all processors. A barrier b placed between i_1 and i_2 in program order has an A-cumulative ordering over m_1 and m_2 when, if m_1 reads from a write w ($i.e.$ $w \xrightarrow{\text{rf}} m_1$), the barrier b imposes a global ordering ($i.e.$ visible to all processors) between w and m_2. This means that all processors see m_2 after w. We will discuss in more details the semantics and use of barriers in Sec. 10.3.3.

Hence the cumulativity of barriers is modeled in our framework by certain sequences $\xrightarrow{\text{rf}};\xrightarrow{\text{po}}$ (resp. $\xrightarrow{\text{po}};\xrightarrow{\text{rf}}$) being global. Formally, we have:

Definition 21 (Properties of Barriers)

$$\mathsf{NC}(\xrightarrow{\text{r}}) \triangleq (\xrightarrow{\text{r}} \subseteq \xrightarrow{\text{po}}) \wedge (\xrightarrow{\text{r}} \subseteq \xrightarrow{\text{ab}}) \textit{ (non-cumulativity)}$$

$$\mathsf{AC}(\xrightarrow{\text{r}}, \xrightarrow{\text{s}}) \triangleq (\xrightarrow{\text{s}} \subseteq \xrightarrow{\text{rf}}) \wedge \big((\xrightarrow{\text{s}};\xrightarrow{\text{r}}) \subseteq \xrightarrow{\text{ab}}\big) \textit{ (A-cumulativity)}$$

$$\mathsf{BC}(\xrightarrow{\text{r}}, \xrightarrow{\text{s}}) \triangleq (\xrightarrow{\text{s}} \subseteq \xrightarrow{\text{rf}}) \wedge \big((\xrightarrow{\text{r}};\xrightarrow{\text{s}}) \subseteq \xrightarrow{\text{ab}}\big) \textit{ (B-cumulativity)}$$

where $\xrightarrow{\text{r}}$ *and* $\xrightarrow{\text{s}}$ *are two relations.*

3.3.4 Architectures

We call a particular model of our class an *architecture*, written A (or A^{ϵ} for A where ab returns the empty relation). We model an architecture by a triple of functions over executions. Hence we consider an architecture as a filter over executions, which determines which executions are valid and which are not. We write ppo (resp. grf, ab, ghb) for the function returning the $\xrightarrow{\text{ppo}}$ (resp. $\xrightarrow{\text{grf}}, \xrightarrow{\text{ab}}$ and $\xrightarrow{\text{ghb}}$) relation $w.r.t.$ A when given an event structure and execution witness:

Definition 22 (Architecture)

$$A \triangleq (\mathsf{ppo}, \mathsf{grf}, \mathsf{ab})$$

We use in the following the record notation $A.f$ for a function f over execution witnesses $w.r.t.$ the architecture A. For example, given an event structure E, an associated execution witness X and two architectures A_1 and A_2, we write $A_1 . \mathsf{ghb}(E, X)$ for the $\xrightarrow{\text{ghb}}$ of the execution (E, X) relative to A_1, while A_2. $\mathsf{ppo}(E,X)$ returns the $\xrightarrow{\text{ppo}}$ of the execution (E,X) relative to A_2. We omit the architecture when it is clear from the context.

Finally, we define $\xrightarrow{\text{ghb}}$ as the union of the global relations:

Definition 23 (Global Happens-Before)

$$\xrightarrow{\text{ghb}} \triangleq \xrightarrow{\text{ppo}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{grf}} \cup \xrightarrow{\text{ab}}$$

3.3.5 Examples

3.3.5.1 Sequential Consistency (SC)

SC (see also Sec. 4.2.1) allows no reordering of events ($\xrightarrow{\text{ppo}}$ equals $\xrightarrow{\text{po}}$ on memory events) and makes writes available to all processors as soon as they are issued ($\xrightarrow{\text{rf}}$ is global, *i.e.* $\xrightarrow{\text{grf}}=\xrightarrow{\text{rf}}$). Thus, there is no need for barriers. We write MM for the function extracting the pairs of memory events in the program order of an event structure, *i.e.* $\text{MM}(E,X) \triangleq \xrightarrow{\text{po}} \cap (\mathbb{M} \times \mathbb{M})$:

Definition 24 (Sequential Consistency)

$$SC \triangleq (\text{MM}, \mathsf{rf}, \lambda(E,X).\emptyset)$$

Thus, the outcome of Fig. 3.1 will never be the result of a *SC* execution. Indeed, the associated execution exhibits the cycle: $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (a)$. Since $\xrightarrow{\text{fr}}$ is always in $\xrightarrow{\text{ghb}}$, and since the program order $\xrightarrow{\text{po}}$ is included in *SC*'s preserved program order, this cycle is a cycle in $\xrightarrow{\text{ghb}}$, hence we contradict the validity of this execution.

3.3.5.2 Sun's Total Store Order (TSO)

TSO (see also Sec. 4.2.2.2) allows two relaxations [AG95]: *write to read program order*, and *read own write early.* The *write to read program order* relaxation means that *TSO*'s preserved program order includes all pairs but the store-load ones. We write RM (resp. WW) for the function extracting the read-read and read-write (resp. write-write) pairs in the program order of an execution witness. Formally, we have $\text{RM}(E,X) \triangleq \xrightarrow{\text{po}} \cap (\mathbb{R} \times \mathbb{M})$ and $\text{WW}(E,X) \triangleq \xrightarrow{\text{po}} \cap \mathbb{W} \times \mathbb{W}$.

The *read own write early* relaxation means that *TSO*'s internal read-from maps are not global, *i.e.* $\xrightarrow{\text{rfi}} \not\subseteq \xrightarrow{\text{ghb}}$. Moreover, *TSO* does not relax the atomicity of stores, *i.e.* $\xrightarrow{\text{rfe}} \subseteq \xrightarrow{\text{ghb}}$. Hence we have (omitting the barriers' semantics):

Definition 25 (TSO^ϵ)

$$\begin{aligned} ppo_{TSO} &\triangleq (\lambda(E,X).\,(\text{RM}(E,X) \cup \text{WW}(E,X)) \\ TSO^\epsilon &\triangleq (ppo_{TSO}, \mathsf{rfe}, \lambda(E,X).\emptyset) \end{aligned}$$

P_0	P_1
$(a)\,\texttt{x} \leftarrow 1$	$(b)\,\texttt{r1} \leftarrow \texttt{x}$
	$(c)\,\texttt{x} \leftarrow 2$

Forbidden: `x=1; r1=1;`

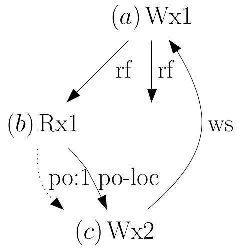


Figure 3.9: Invalid Execution According to the $\mathsf{uniproc}$ Criterion

Thus, the outcome of Fig. 3.1 can be the result of a *TSO* execution. Even if the associated execution (E, X) exhibits the cycle $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (a)$, this does not form a cycle in $\mathit{TSO}\,.\,\mathsf{ghb}(E, X)$. Indeed, the write-read pairs in $(a) \xrightarrow{\text{po}} (b)$ on P_0 and $(c) \xrightarrow{\text{po}} (d)$ on P_1 are not included in *TSO*'s program order, hence can be reordered.

3.4 Validity of an Execution

We now add two sanity conditions to the above.

3.4.1 Uniprocessor Behaviour

First, we require each processor to respect memory coherence for each location [CLS] (*i.e.* the per-location write serialisation): if a processor writes *e.g.* v to ℓ and then reads v' from ℓ, then the associated writes w and w' should be in this order in the write serialisation, *i.e.* w' should not precede w in the write serialisation. We formalise this notion as follows.

3.4.1.1 Definition

We define the relation $\xrightarrow{\text{po-loc}}$ over accesses to the same location in the program order:

$$m_1 \xrightarrow{\text{po-loc}} m_2 \triangleq m_1 \xrightarrow{\text{po}} m_2 \wedge \mathsf{loc}(m_1) = \mathsf{loc}(m_2)$$

We require $\xrightarrow{\text{po-loc}}$ to be compatible with $\xrightarrow{\text{com}}$ (*i.e.* $\xrightarrow{\text{rf}} \cup \xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}}$):

Definition 26 (Uniprocessor Check)

$$\mathsf{uniproc}(E, X) \triangleq \mathsf{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}})$$

For example, in Fig. 3.9, we have $(c) \xrightarrow{\text{ws}} (a)$ (by x final value) and $(a) \xrightarrow{\text{rf}} (b)$ (by `r1` final value). The cycle $(a) \xrightarrow{\text{rf}} (b) \xrightarrow{\text{po-loc}} (c) \xrightarrow{\text{ws}} (a)$ invalidates this execution: (b) cannot read from (a) as it is a future value of x in $\xrightarrow{\text{ws}}$.

As a side note, the uniproc check corresponds, as we shall see in Sec. 4.2.1, to checking that SC holds per location.

3.4.1.2 Alternative Formulations

We give here two alternative formulations of the uniproc definition, which we show equivalent with each other. We omit these equivalence proofs in this chapter to ease the reading but they can be found in Chap. A.

We show easily that the transitive closure of $\xrightarrow{\text{com}}$, written $(\xrightarrow{\text{com}})^+$, is equal to $\xrightarrow{\text{com}} \cup (\xrightarrow{\text{ws}}; \xrightarrow{\text{rf}}) \cup (\xrightarrow{\text{fr}}; \xrightarrow{\text{rf}})$.

We write $x \overset{\text{hat}}{\leftrightarrow} y$ when x and y are both reads, reading from the same write. Formally, we have (since this relation is symmetric, we use a double arrow notation):

Definition 27 (Hat Relation)

$$x \overset{\text{hat}}{\leftrightarrow} y \;\triangleq\; \exists w, w \xrightarrow{\text{rf}} x \wedge w \xrightarrow{\text{rf}} y$$

Let us consider these three formulations of uniproc:

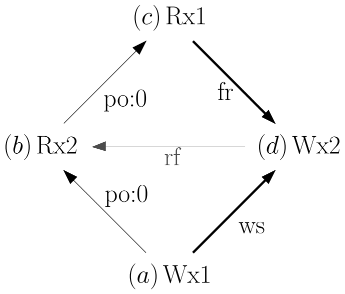
$$\begin{array}{lc}
(\mathit{Uni1}) & \mathsf{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}}) \\
(\mathit{Uni2}) & \forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow \neg(y\,(\xrightarrow{\text{com}})^+\,x) \\
(\mathit{Uni3}) & \xrightarrow{\text{po-loc}} \subseteq ((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow})
\end{array}$$

We chose the first one because it is synthetic, but its intuition arises more clearly in $(\mathit{Uni2})$ (and equivalently in $(\mathit{Uni3})$). The $(\mathit{Uni2})$ check requires indeed that if two events x and y are in $\xrightarrow{\text{po-loc}}$, *i.e.* in program order and to the same location, there is no path in the communication arrows $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{ws}}$ or $\xrightarrow{\text{fr}}$ leading from y to x. This means that two events relative to the same location on the same processor cannot be seen in the converse order by other processors.

As a remark, it is interesting to note that the uniproc check can spare the cost of including certains pairs of events in program order in the preserved program order of an architecture. Consider for example two writes to the same location in program order. They are necessarily (by uniproc) included in $\xrightarrow{\text{ghb}}$. Indeed, such a pair is in $\xrightarrow{\text{po-loc}}$, thus in $((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow})$ by $(\mathit{Uni3})$. Moreover, we know that $(\xrightarrow{\text{com}})^+$ is equal to $(\xrightarrow{\text{com}} \cup (\xrightarrow{\text{ws}}; \xrightarrow{\text{rf}}) \cup (\xrightarrow{\text{fr}}; \xrightarrow{\text{rf}}))$. Hence, a write-write pair to the same location is, by $(\mathit{Uni3})$, in $(\xrightarrow{\text{com}} \cup (\xrightarrow{\text{ws}}; \xrightarrow{\text{rf}}) \cup (\xrightarrow{\text{fr}}; \xrightarrow{\text{rf}}))$. The cases $(\xrightarrow{\text{ws}}; \xrightarrow{\text{rf}})$ and $(\xrightarrow{\text{fr}}; \xrightarrow{\text{rf}})$ do not apply here because of the directions of the events. Hence a write-write pair to the same location is in $\xrightarrow{\text{com}}$, *i.e.* in $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{rf}} \cup \xrightarrow{\text{fr}}$. The cases $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{fr}}$ do not apply because of the directions of the events, hence such a pair is in $\xrightarrow{\text{ws}}$. We know, by hypothesis of our framework, that $\xrightarrow{\text{ws}}$ is always global. Hence, there is no need to specify

P_0	P_1
(a) `x` $\leftarrow$ `1`	(d) `x` $\leftarrow$ `2`
(b) `r1` $\leftarrow$ `x`	
(c) `r2` $\leftarrow$ `x`	
Observed? `r1=2; r2=1; x=2`	

(a) Program



(b) An Execution Witness

Figure 3.10: Load-Load Hasard Example

write-write pairs to the same location in the preserved program order, since we know that they are preserved globally (*i.e.* in $\xrightarrow{\text{ghb}}$) by the uniproc check.

The same reasoning applies for read-write pairs to the same location: such pairs are necessarily in $\xrightarrow{\text{fr}}$, thus in $\xrightarrow{\text{ghb}}$.

Hence, the uniproc check can be viewed as a minimal condition imposed by a machine: the write-write and read-write pairs to the same location in the program order are necessarily preserved globally in the order specified by the program.

3.4.1.3 Load-Load Hasard

As a side note, all the models embraced by our framework agree with the uniproc check, except Sun *RMO* [spa94a] (see also Sec. 4.2.2.4). *RMO* preserves the program order between the write-write and read-write pairs to the same location, and the read-read and read-write pairs in *dependency* [spa94a].

We postulate a $\xrightarrow{\text{dp}}$ relation to model the dependencies between instructions, such as *data* or *control dependencies* [pow09, pp. 653-668]. For example in PowerPC, consider a load `lwz r5,0(r1)` followed in program order by a load `lwzx r6,r5,r2`. The second load is indexed by the register `r5`; this register is used as a target by the preceding load in program order. Hence in PowerPC, the second load depends on the first one. The $\xrightarrow{\text{dp}}$ relation is a subrelation of $\xrightarrow{\text{po}}$, and always has a read at its source.

We write $\mathsf{dp}(E, X)$ for the pairs in dependency in an execution (E, X). We write $\mathrm{M}_\ell\mathrm{W}_\ell(E, X)$ for the read-write and write-write pairs to the same location ℓ, in an execution X. *RMO* has the same store buffering and store atomicity policy as *TSO*, hence (omitting the barriers' semantics):

P_0	P_1
(a) `r1` ← `x`	(c) `r4` ← `y`
`r9` ← `xor r1,r1`	`r9` ← `xor r4,r4`
(b) `y` ← `1+r9`	(d) `x` ← `1+r9`

Forbidden: `r1=1`; `r4=1`;

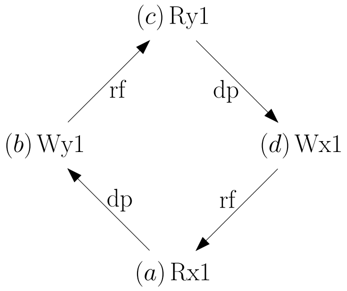


Figure 3.11: Invalid Execution According to the thin Criterion

Definition 28 (RMO^{ϵ})

$$\begin{aligned} ppo_{RMO} &\triangleq \lambda(E,X).(\mathsf{dp}(E,X) \cup \bigcup_{\ell} \mathrm{M}_{\ell}\mathrm{W}_{\ell}(E,X)) \\ RMO^{\epsilon} &\triangleq (ppo_{RMO}, \mathsf{rfe}, \lambda(E,X).\emptyset) \end{aligned}$$

RMO allows indeed *load-load hasard*, meaning two reads from the same location in program order may be reordered. We give in Fig. 3.10 an example program of load-load hasard: the read (b) from x and the read (c) from x on P_0 have been reordered. The read (c) reads from the write (a) on P_0, hence $(a) \xrightarrow{\text{rf}} (c)$, which is not depicted in Fig. 3.10 to ease the reading. The read (b) reads from the write (d) on P_1, hence $(d) \xrightarrow{\text{rf}} (b)$. Suppose $(a) \xrightarrow{\text{ws}} (d)$, then, since $(a) \xrightarrow{\text{rf}} (c)$, we have $(c) \xrightarrow{\text{fr}} (d)$. Since we have $(b) \xrightarrow{\text{po}} (c)$, we exhibit a cycle which is a contradiction to uniproc: $(b) \xrightarrow{\text{po}} (c) \xrightarrow{\text{fr}} (d) \xrightarrow{\text{rf}} (b)$. The program order between the two reads (b) and (c) on P_0 is not respected, even though they are to the same location x.

To allow load-load hasard, we define the relation $\xrightarrow{\text{llh-po-loc}}$ over accesses to the same location in the program order as $\xrightarrow{\text{po-loc}}$ except for read-read pairs:

$$m_1 \xrightarrow{\text{llh-po-loc}} m_2 \triangleq m_1 \xrightarrow{\text{po}} m_2 \wedge \mathsf{loc}(m_1) = \mathsf{loc}(m_2) \wedge \neg(m_1 \in \mathbb{R} \wedge m_2 \in \mathbb{R})$$

Then we slightly alter the definition of uniproc to:

Definition 29 (Load-Load Hasard Uniproc)

$$\mathsf{llh\text{-}uniproc}(E,X) \triangleq \mathsf{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{llh-po-loc}})$$

3.4.2 Thin Air

Second, we rule out programs where values come *out of thin air* [MPA]. This means that we forbid the *causal loops*, as illustrated in Fig. 3.11. In this example, the write (b) to y on P_0 is dependent on the read (a) from x on P_0, because the `xor` instruction between them does a calculation on the value written by (a) in `r1`, and writes the result into `r9`, later used by (b). Similarly on P_1, (c) and (d) are dependent. Suppose the read (a) from x on P_0 reads from the write (d) to x on P_1, and similarly the read (c) from y on P_1 reads from the write (b) to y on P_0, as depicted by the execution in Fig. 3.11. In this case, the values read by (a) and (c) seem to come out of thin air, because they cannot be determined.

The definition of this check is parameterised by the definition of dependencies, relative to the architecture:

Definition 30 (Thin Air Check)

$$\mathsf{thin}(\mathsf{dp}, E, X) \triangleq \mathsf{acyclic}(\mathsf{rf}(X) \cup \mathsf{dp}(E, X))$$

This check is directly inspired of Alpha's documentation [alp02, (I) 5-15, p. 245]. Alpha is indeed the only model embraced by our framework that requires this check. *Alpha* maintains the write-write, read-read and read-write pairs to the same location [alp02]. We write $\mathrm{R}_\ell\mathrm{M}_\ell(E, X)$ (resp. $\mathrm{W}_\ell\mathrm{W}_\ell(E, X)$) for the read-read and read-write (resp. write-write) pairs to the same location ℓ in an execution (E, X). Similarly to the Sun models, Alpha's external $\xrightarrow{\text{rf}}$ are global whereas internal $\xrightarrow{\text{rf}}$ are not (we omit the barriers' semantics):

Definition 31 (*Alpha*$^\epsilon$)

$$\begin{aligned} \mathit{ppo}_{\mathit{Alpha}} &\triangleq \lambda(E, X) . (\textstyle\bigcup_\ell \mathrm{R}_\ell\mathrm{M}_\ell(E, X) \cup \bigcup_\ell \mathrm{W}_\ell\mathrm{W}_\ell(E, X)) \\ \mathit{Alpha}^\epsilon &\triangleq (\mathit{ppo}_{\mathit{Alpha}}, \mathsf{rfe}, \lambda(E, X) . \emptyset) \end{aligned}$$

All the other models we present have indeed a sufficiently large $\xrightarrow{\text{ppo}}$ that includes $\xrightarrow{\text{dp}}$, which removes the necessity of the thin check. As the preserved program order of Alpha (see also Sec. 4.2.3) does not include $\xrightarrow{\text{dp}}$, and since Alpha explicitely prevents causal loops, this additional check is required.

3.4.3 Validity

We define the validity of an execution *w.r.t.* an architecture A as the conjunction of four checks. The first three, namely $\mathsf{wf}(X)$, $\mathsf{uniproc}(E, X)$ and $\mathsf{thin}(E, X)$ are independent from the architecture. The last one, *i.e.* $A . \mathsf{ghb}(E, X)$, characterises the architecture. We write $A . \mathsf{valid}(E, X)$ when the execution (E, X) is valid on the architecture A, and $A . \mathsf{ghb}(E, X)$ for the $\xrightarrow{\text{ghb}}$ induced by A on (E, X):

Definition 32 (Validity)

$$A . \mathsf{valid}(E, X) \triangleq \mathsf{wf}(X) \wedge \mathsf{uniproc}(E, X) \wedge \mathsf{thin}(E, X) \wedge \mathsf{acyclic}(A . \mathsf{ghb}(E, X))$$

For example, the execution of Fig. 3.2 is invalid on SC. Indeed the $SC\,.\,\mathsf{ghb}(E,X)$ of this execution contains $\overset{\mathrm{po}}{\rightarrow}$ and $\overset{\mathrm{fr}}{\rightarrow}$, therefore has a cycle: $(a) \overset{\mathrm{po}}{\rightarrow} (b) \overset{\mathrm{fr}}{\rightarrow} (c) \overset{\mathrm{po}}{\rightarrow} (d) \overset{\mathrm{fr}}{\rightarrow} (a)$. On the contrary, the $TSO\,.\,\mathsf{ghb}(E,X)$ of this execution does not contain any $\overset{\mathrm{po}}{\rightarrow}$ arrow which source is a write and target a read, hence does not contain $(a) \overset{\mathrm{po}}{\rightarrow} (b)$ and $(c) \overset{\mathrm{po}}{\rightarrow} (d)$. Thus, there is no cycle in $TSO\,.\,\mathsf{ghb}(E,X)$, which means that this execution is not forbidden on TSO.

3.5 Comparing Architectures

From our definition of architecture arises a simple notion of comparison amongst them. $A_1 \leq A_2$ means that A_1 is *weaker* than A_2:

Definition 33 (Weaker)

$$A_1 \leq A_2 \triangleq (\overset{\mathrm{ppo}_1}{\rightarrow} \subseteq \overset{\mathrm{ppo}_2}{\rightarrow}) \wedge (\overset{\mathrm{grf}_1}{\rightarrow} \subseteq \overset{\mathrm{grf}_2}{\rightarrow})$$

As an example, TSO^{ϵ} is weaker than SC.

3.5.1 Validity Is Decreasing

The validity of an execution is decreasing *w.r.t.* the strength of the predicate; *i.e.* a weak architecture exhibits at least all the behaviours of a stronger one:

Theorem 1 (Validity Is Decreasing)

$$\forall A_1 A_2,\ (A_1 \leq A_2) \Rightarrow (\forall EX, A_2^{\epsilon}\,.\,\mathsf{valid}(E,X) \Rightarrow A_1^{\epsilon}\,.\,\mathsf{valid}(E,X))$$

Proof *From $A_1 \leq A_2$, we immediately have $A_1^{\epsilon}\,.\,\mathsf{ghb} \subseteq A_2^{\epsilon}\,.\,\mathsf{ghb}$, thus if $A_2^{\epsilon}\,.\,\mathsf{ghb}$ is acyclic, so is $A_1^{\epsilon}\,.\,\mathsf{ghb}$.* □

For example, since TSO^{ϵ} is weaker than SC, all the executions valid on SC are valid on TSO.

3.5.2 Monotonicity of Validity

The converse is not always true. However, some programs running on an architecture A_1^{ϵ} exhibit executions that would be valid on a stronger architecture A_2^{ϵ}. We characterise all such executions as follows, where the expression $A_1\,.\,\mathsf{check}_{\mathsf{A_2}}(E,X)$ means that the execution X, while running on the weak architecture A_1, would also be valid on the stronger architecture A_2:

Definition 34 (Strong Execution On Weak Architecture)

$$A_1\,.\,\mathsf{check}_{\mathsf{A_2}}(E,X) \ \triangleq\ \mathsf{acyclic}(\overset{\mathrm{grf}_2}{\rightarrow} \cup \overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{ppo}_2}{\rightarrow})$$

We show that executions satisfying this criterion are valid on A_1^ϵ if and only if they are valid on A_2^ϵ:

Theorem 2 (Characterisation)

$$\begin{gathered}\forall A_1 A_2,\ (A_1 \leq A_2) \Rightarrow \\ (\forall EX, (A_1^\epsilon\,.\,\mathsf{valid}(E,X) \wedge A_1\,.\,\mathsf{check}_{\mathsf{A_2}}(E,X)) \Leftrightarrow A_2^\epsilon\,.\,\mathsf{valid}(E,X))\end{gathered}$$

Proof

- $\Rightarrow$ (E,X) *being valid on* A_1^ϵ*, we have all requirements—well-formedness,* uniproc *and* thin*—to guarantee* (E,X) *is valid on* A_2^ϵ*, except* $A_2^\epsilon\,.\,\mathsf{valid}(E,X)$*, which holds by the hypothesis* $\mathsf{check}_{\mathsf{A_2}}$.
- $\Leftarrow$ (E,X) *being valid on* A_2^ϵ *gives us all requirements—well-formedness,* uniproc *and* thin*—to guarantee its validity on* A_1^ϵ *except the last one* $A_1^\epsilon\,.\,\mathsf{valid}(E,X)$*. As* $A_1 \leq A_2$*, we know that* $A_1^\epsilon\,.\,\mathsf{ghb} \subseteq A_2^\epsilon\,.\,\mathsf{ghb}$*, thus the acyclicity requirement for* $A_1^\epsilon\,.\,\mathsf{ghb}$ *holds if* $A_2^\epsilon\,.\,\mathsf{ghb}$ *is acyclic.* □

For example, consider the execution of the test of Fig. 3.1(a) where P_0 executes its instructions before P_1 does: $(a) \overset{\text{po}}{\rightarrow} (b) \overset{\text{fr}}{\rightarrow} (c) \overset{\text{po}}{\rightarrow} (d)$ and $(a) \overset{\text{rf}}{\rightarrow} (d)$. It is valid on TSO since there is no cycle in $TSO\,.\,\mathsf{ghb}(E,X)$. It also satisfies $TSO\,.\,\mathsf{check}_{SC}(E,X)$ since there is no cycle in $SC\,.\,\mathsf{ghb}(E,X)$. Hence it is valid on SC as well.

These theorems, though fairly simple, will be useful to compare two models and to restore a strong model from a weaker one, as we do in Chap. 11.

Chapter 4

Classical Models

We expose here how we implement several classical models in our framework, namely Sequential Consistency (SC) [Lam79], the Sparc hierarchy (*i.e.* TSO, PSO and RMO [spa94a]) and Alpha [alp02]. We prove our implementations equivalent to the original definitions. We present the models from the stronger (*w.r.t.* the order $\leq$), namely SC, to the weaker, namely Alpha. We show in Fig. 4.1 the inclusion of these models *w.r.t.* each other. The inclusion is here in terms of the behaviours each model authorises, therefore is in the converse order that the order $\leq$ induces, as expressed by Thm. 1.

4.1 Implementing an Architecture

We present each of the following native models in terms of an order $\overset{\text{ex}}{\rightarrow}$ representing the order in which the events are globally performed.

All the native definitions of the models we present here roughly follow the same generic form. In these definitions, an execution $\overset{\text{ex}}{\rightarrow}$ is valid if it is an order on events which contains a certain subrelation of the program order. This relation corresponds to our preserved program order.

The order $\overset{\text{ex}}{\rightarrow}$ is defined as partial in Alpha's documentation [alp02], or

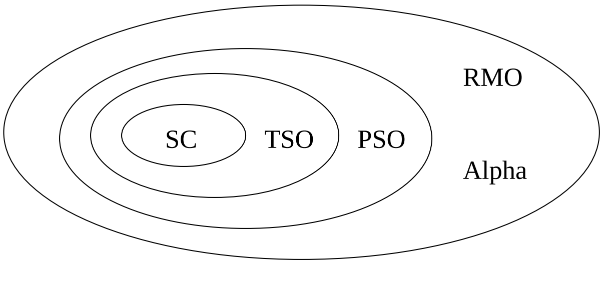


Figure 4.1: Inclusion of Some Architectures

early versions of the Sparc documentation [spa92]. In the current version of Sparc documentation [spa94b], it is defined as a total order. By Ax. 1, any partial order can be extended to a total order. Hence we will define the native versions of the models in terms of a total order. This means for example that for an execution defined as a partial order, we consider all its linear extensions to be valid native executions.

For a given architecture A, we write $A\,.\,\mathsf{native}(\xrightarrow{\text{ex}})$ when $\xrightarrow{\text{ex}}$ satisfies the conditions imposed by A. Formally, we have:

Definition 35 (Native Definition of an Architecture)

$$A\,.\,\mathsf{native}(E, \xrightarrow{\text{ex}}) \;\triangleq\; \mathsf{total\text{-}order}(\xrightarrow{\text{ex}}, \mathsf{evts}(E)) \wedge (A\,.\,\mathsf{ppo}(E) \subseteq \xrightarrow{\text{ex}})$$

Let A be an architecture of our framework. We want to show that the validity of an execution on A corresponds to the definition above. This means that whenever an execution $\xrightarrow{\text{ex}}$ is valid on A according to the definition above, we can build an execution witness (E, X) which is valid on A, such that $\xrightarrow{\text{ex}}$ and (E, X) have the same events and the same communication relations. Conversely, from an execution (E, X) valid on A, we are able to build an execution $\xrightarrow{\text{ex}}$ valid on A with the same events and communication relations. Formally, writing $A\,.\,\mathsf{wit}(\xrightarrow{\text{ex}})$ for the execution witness built from $\xrightarrow{\text{ex}}$, we would like to show that:

Goal 1 (Equivalence of validity on A)

$$\forall EX,\; A\,.\,\mathsf{valid}(E, X) \Leftrightarrow \exists \xrightarrow{\text{ex}},\; A\,.\,\mathsf{native}(E, \xrightarrow{\text{ex}}) \wedge A\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

4.1.1 Building an Execution Witness From an Order

We consider two relations to be *compatible* when their union is acyclic. Consider an architecture A without barriers, *i.e.* $\xrightarrow{\text{ab}_A} = \emptyset$. The goal above means in particular that, for any event structure E, one can build an execution, associated with E and valid on A, from a total order $\xrightarrow{\text{o}}$ on $\mathsf{evts}(E)$ compatible with $A\,.\,\mathsf{ppo}(E)$.

Consider *e.g.* the event structure $(\{(a), (b), (c), (d)\}, \{(a) \xrightarrow{\text{po}} (b), (c) \xrightarrow{\text{po}} (d)\})$ associated with the program of Fig. 4.2(a). On SC we have $(a) \xrightarrow{\text{ppo}} (b)$ and $(c) \xrightarrow{\text{ppo}} (d)$. Hence we can build a valid SC execution from the order $(a) \xrightarrow{\text{o}} (b) \xrightarrow{\text{o}} (c) \xrightarrow{\text{o}} (d)$, which is the one we give in Fig. 4.3. The first write in the order $\xrightarrow{\text{o}}$ is (b), a write to y, which is immediately followed by the read (c) to y, hence we have $(b) \xrightarrow{\text{rf}} (c)$. There is no write preceding the read (a) from x, hence (a) reads from the initial state. Moreover, this initial write to x precedes the write (d) in $\xrightarrow{\text{ws}}$, hence $(a) \xrightarrow{\text{fr}} (d)$.

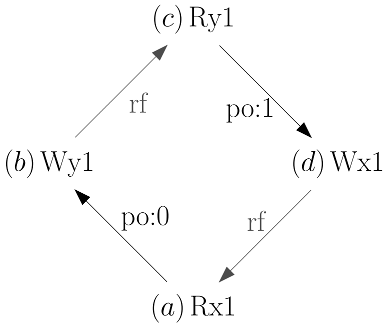
We need to build an execution witness from a given order $\xrightarrow{\text{ex}}$. In order to do so, we need to extract $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ from an order $\xrightarrow{\text{ex}}$. We write $\mathsf{rf}(\xrightarrow{\text{ex}})$ (resp. $\mathsf{ws}(\xrightarrow{\text{ex}})$) for the $\xrightarrow{\text{rf}}$ (resp. $\xrightarrow{\text{ws}}$) extracted from $\xrightarrow{\text{ex}}$. We have $(x, y) \in \mathsf{rf}(\xrightarrow{\text{ex}})$ when

Init: `x=0`; `y=0`; `z=0`

P_0	P_1
(a) `r1` ← `x`	(c) `r2` ← `y`
(b) `y` ← `1`	(d) `x` ← `1`

Observed? `r1=1`; `r2=1`;

(a) A program



(b) A non-SC execution

Figure 4.2: A program and a non-SC execution

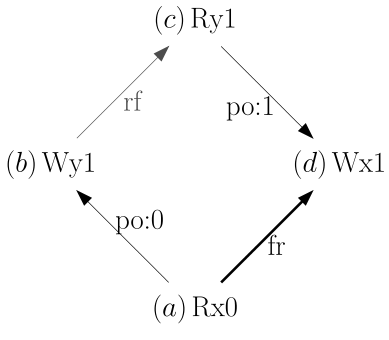


Figure 4.3: An SC execution for the test of Fig. 4.2(a)

x is a write and y a read, both to the same location, such that x is a maximal previous write to this location before y in $\xrightarrow{\text{ex}}$. We have $(x, y) \in \mathsf{ws}(\xrightarrow{\text{ex}})$ when x and y are writes to the same location and $x \xrightarrow{\text{ex}} y$. Formally, writing $\mathsf{pw}(\xrightarrow{\text{ex}}, r)$ for the set of writes to the same location that precede the read event r in an order $\xrightarrow{\text{ex}}$, we extract our $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ relations from $\xrightarrow{\text{ex}}$ as follows:

Definition 36 (Extraction of $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ From an Order $\xrightarrow{\text{ex}}$)

$$\begin{array}{rcl}
\mathsf{pw}(\xrightarrow{\text{ex}}, r) & \triangleq & \{w \mid \mathsf{loc}(w) = \mathsf{loc}(r) \wedge w \xrightarrow{\text{ex}} r\} \\
\mathsf{rf}(\xrightarrow{\text{ex}}) & \triangleq & \{(w, r) \mid w = \max_{\xrightarrow{\text{ex}}} \left(\mathsf{pw}((\xrightarrow{\text{ex}}), r)\right)\} \\
\mathsf{ws}(\xrightarrow{\text{ex}}) & \triangleq & \bigcup_{\ell} (\mathbb{W}_\ell \times \mathbb{W}_\ell) \cap \xrightarrow{\text{ex}}
\end{array}$$

We derive the from-read map as in Sec. 3.2.3:

Definition 37 (Extracted $\xrightarrow{\text{fr}}$)

$$(r, w) \in \mathsf{fr}(\xrightarrow{\text{ex}}) \;\triangleq\; \exists w', (w', r) \in \mathsf{rf}(\xrightarrow{\text{ex}}) \wedge (w', w) \in \mathsf{ws}(\xrightarrow{\text{ex}})$$

We show that the extracted read-from maps $\mathsf{rf}(\xrightarrow{\text{ex}})$, write serialisation $\mathsf{ws}(\xrightarrow{\text{ex}})$ and from-read maps $\mathsf{fr}(\xrightarrow{\text{ex}})$ are included in $\xrightarrow{\text{ex}}$:

Lemma 6 (Inclusion of Extracted Communication In $\xrightarrow{\text{ex}}$)

$$\forall \xrightarrow{\text{ex}}, \mathsf{rf}(\xrightarrow{\text{ex}}) \subseteq \xrightarrow{\text{ex}} \wedge \mathsf{ws}(\xrightarrow{\text{ex}}) \subseteq \xrightarrow{\text{ex}} \wedge \mathsf{fr}(\xrightarrow{\text{ex}}) \subseteq \xrightarrow{\text{ex}}$$

Proof

- *Inclusion of read-from maps: The read-from maps extracted from $\xrightarrow{\text{ex}}$ are by definition in $\xrightarrow{\text{ex}}$.*

- *Inclusion of write serialisation: The write serialisation extracted from $\xrightarrow{\text{ex}}$ is included in $\xrightarrow{\text{ex}}$ since it is by definition built from an intersection with $\xrightarrow{\text{ex}}$.*

- *Inclusion of from-read maps: Consider two events x and y such that $(x, y) \in \mathsf{fr}(\xrightarrow{\text{ex}})$. We want to show that $x \xrightarrow{\text{ex}} y$. Since $\xrightarrow{\text{ex}}$ is a total order, we know that either $x \xrightarrow{\text{ex}} y$, in which case we have the result, or $y \xrightarrow{\text{ex}} x$. Suppose this last possibility, i.e. $y \xrightarrow{\text{ex}} x$. We know that y is a write to x's location, since it is the target of a $\xrightarrow{\text{fr}}$ which source is x. Therefore, if $y \xrightarrow{\text{ex}} x$, we know that y is a previous write to x in $\xrightarrow{\text{ex}}$. Hence we have $y \in \mathsf{pw}(\xrightarrow{\text{ex}}, x)$. Moreover, since $(x, y) \in \mathsf{fr}(\xrightarrow{\text{ex}})$, we know by definition that there exists w_x such that $(w_x, x) \in \mathsf{rf}(\xrightarrow{\text{ex}})$ and $(w_x, y) \in \mathsf{ws}(\xrightarrow{\text{ex}})$. Since $\mathsf{ws}(\xrightarrow{\text{ex}})$ is included in $\xrightarrow{\text{ex}}$, we have $w_x \xrightarrow{\text{ex}} y$. But by definition of $\mathsf{rf}(\xrightarrow{\text{ex}})$, and since $(w_x, x) \in \mathsf{rf}(\xrightarrow{\text{ex}})$, w_x is the maximal previous write to x in $\xrightarrow{\text{ex}}$. Since $w_x \xrightarrow{\text{ex}} y$ and y is also a previous write, this contradicts the maximality of w_x.* □

4.1.2 Sketch of Proof

4.1.2.1 From the native definition to ours

The extracted $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ are well formed in a finite execution, hence an execution witness built from these relations is well formed as well. If these $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ satisfy the uniproc and thin checks, then the execution witness built out of $\xrightarrow{\text{ex}}$ is valid on a given architecture A.

We show that an extracted execution witness respects the uniproc check:

Lemma 7 (Extracted Execution Witness Respects uniproc)

$$\begin{gathered}\forall A, \mathbb{E}, \xrightarrow{\text{po}}, \xrightarrow{\text{ex}}, X,\\ \mathsf{total\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge X = A.\,\mathsf{wit}(\xrightarrow{\text{ex}}) \wedge\\ \mathsf{acyclic}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}}) \Rightarrow \mathsf{uniproc}(X)\end{gathered}$$

Proof *We know by Lem. 29 (see App. A) that the* uniproc *check* (*Uni1*) *implies* (*Uni2*), *i.e.* $\forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow \neg(y\,(\xrightarrow{\text{com}})^+\,x)$. *Let us suppose as a contradiction two events x and y such that $x \xrightarrow{\text{po-loc}} y$ and $y\,(\xrightarrow{\text{com}})^+\,x$:*

- *if $(y, x) \in \mathsf{rf}(\xrightarrow{\text{ex}})$, we have $y \xrightarrow{\text{ex}} x$ by Lem. 6. Therefore, since $x \xrightarrow{\text{po-loc}} y$, we have a cycle in $\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}}$, a contradiction.*
- *if $(y, x) \in \mathsf{ws}(\xrightarrow{\text{ex}})$, we have $y \xrightarrow{\text{ex}} x$ by Lem. 6. Since $x \xrightarrow{\text{po-loc}} y$, we have a cycle in $\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}}$, a contradiction.*
- *if $(y, x) \in \mathsf{fr}(\xrightarrow{\text{ex}})$, there exists w_y such that $(w_y, y) \in \mathsf{rf}(\xrightarrow{\text{ex}})$ and $(w_y, x) \in \mathsf{ws}(\xrightarrow{\text{ex}})$. Therefore x is a previous write to y's location that occurs after w_y in $\xrightarrow{\text{ws}}$ thus in $\xrightarrow{\text{ex}}$. This contradicts $(w_y, y) \in \mathsf{rf}(\xrightarrow{\text{ex}})$, i.e. the maximality of w_y in the set of previous writes to y.*
- *if $y \xrightarrow{\text{ws}};\xrightarrow{\text{rf}} x$, there exists w_x such that $y \xrightarrow{\text{ws}} w_x \xrightarrow{\text{rf}} x$. Since $w_x \xrightarrow{\text{rf}} x$, we have $w_x \xrightarrow{\text{ex}} x$ by Lem. 6. By the same lemma, we have $y \xrightarrow{\text{ex}} w_x$. Hence we have a cycle in $\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}}$: $y \xrightarrow{\text{ex}} w_x \xrightarrow{\text{ex}} x \xrightarrow{\text{po-loc}} y$, a contradiction.*
- *if $y \xrightarrow{\text{fr}};\xrightarrow{\text{rf}} x$, there exists w_x such that $y \xrightarrow{\text{fr}} w_x \xrightarrow{\text{rf}} x$. Since $w_x \xrightarrow{\text{rf}} x$, we have $w_x \xrightarrow{\text{ex}} x$ by Lem. 6. By the same lemma, we have $y \xrightarrow{\text{ex}} w_x$. Hence we have a cycle in $\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}}$: $y \xrightarrow{\text{ex}} w_x \xrightarrow{\text{ex}} x \xrightarrow{\text{po-loc}} y$, a contradiction.* □

Lemma 8 (Validity of Extracted Execution Witness)

$$\begin{gathered}\forall A, \mathbb{E}, \xrightarrow{\text{po}}, \xrightarrow{\text{ex}}, \xrightarrow{\text{dp}}, X,\\ \mathsf{total\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge X = A.\,\mathsf{wit}(\xrightarrow{\text{ex}}) \wedge\\ \mathsf{acyclic}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}}) \wedge \mathsf{acyclic}(\mathsf{rf}(\xrightarrow{\text{ex}}) \cup \xrightarrow{\text{dp}}) \wedge\\ \mathsf{acyclic}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{ppo}_A}) \Rightarrow A.\,\mathsf{valid}(X)\end{gathered}$$

Proof *By Sec. 3.4.3, X is valid on A if X is well formed, respects the* uniproc *and* thin *checks, and $A.\,\mathsf{ghb}(X)$ is acyclic.*

- *Well-formedness:* $\mathsf{rf}(\xrightarrow{\text{ex}})$ *and* $\mathsf{ws}(\xrightarrow{\text{ex}})$ *are trivially well formed, hence* $X = (\mathbb{E}, \xrightarrow{\text{po}}, \mathsf{rf}(\xrightarrow{\text{ex}}), \mathsf{ws}(\xrightarrow{\text{ex}}))$ *is well formed as well.*

- *Uniproc: we want to show that* X *respects the* uniproc *check. Since we know by hypothesis that* $\mathsf{acyclic}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po-loc}})$*, Lem. 6 applies directly.*

- *Thin: we want to show that* X *respects the* thin *check, i.e. that* $\mathsf{acyclic}(\mathsf{rf}(\xrightarrow{\text{ex}}) \cup \xrightarrow{\text{dp}})$*. This is trivial by hypothesis.*

- *Acyclicity of* $\xrightarrow{\text{ghb}}$*: we want to show that* $A.\,\mathsf{ghb}(X)$ *is acyclic. We have by definition that* $\mathsf{ghb}(X) = (\mathsf{grf}(\xrightarrow{\text{o}}) \cup \mathsf{ws}(\xrightarrow{\text{o}}) \cup \mathsf{fr}(\xrightarrow{\text{o}}) \cup \xrightarrow{\text{ppo}_A})$*. Moreover, we know that* $\mathsf{grf}(\xrightarrow{\text{o}})$*,* $\mathsf{ws}(\xrightarrow{\text{o}})$ *and* $\mathsf{fr}(\xrightarrow{\text{o}})$ *are included in* $\xrightarrow{\text{o}}$ *by Lem. 6, therefore we have* $\mathsf{ghb}(X) \subseteq \xrightarrow{\text{o}} \cup \xrightarrow{\text{ppo}_A}$*. By hypothesis, we know that* $\mathsf{acyclic}(\xrightarrow{\text{o}} \cup \xrightarrow{\text{ppo}_A})$*. Since* $A.\,\mathsf{ghb}(X)$ *is included in this relation, this entails its acyclicity.* □

This is enough to show that the native validity entails our definition of validity, for a given architecture A.

4.1.2.2 From our implementation to the native one

Conversely, to show that one of our execution witnesses corresponds to an execution of the native model, we build an order which satisfies the axioms of the native model. This order will typically be the $\xrightarrow{\text{ghb}}$ of our execution witness, or more precisely a linear extension of it, so as to build a total order. Formally, we want to show:

Lemma 9 (From A to its native definition)

$$\forall EX,\ A.\,\mathsf{valid}(E,X) \Rightarrow \exists \xrightarrow{\text{ex}},\ A.\,\mathsf{native}(E, \xrightarrow{\text{ex}}) \wedge A.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

Proof *From* (E,X) *being valid on* A*, we have* $\mathsf{acyclic}(A.\,\mathsf{ghb}(E,X))$*. Therefore any linear extension* $\xrightarrow{\text{ex}}$ *of* $A.\,\mathsf{ghb}(E,X)$ *is a total order on* $\mathbb{E}$ *which contains* $A.\,\mathsf{ppo}(E)$*, since by definition* $\xrightarrow{\text{ppo}}$ *is included in* $\xrightarrow{\text{ghb}}$*. Hence* $\xrightarrow{\text{ex}}$ *is a total order on the events of* (E,X) *which contains* $\xrightarrow{\text{ppo}}$*. Thus* $\xrightarrow{\text{ex}}$ *is such that* $A.\,\mathsf{native}(E, \xrightarrow{\text{ex}})$*.*

Let us show that $A.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$*, i.e.* $\mathsf{ws}(\xrightarrow{\text{ex}}) = \mathsf{ws}(X)$ *and* $\mathsf{rf}(\xrightarrow{\text{ex}}) = \mathsf{rf}(X)$*.*

- *Equality of write serialisations: by definition, we know that* $\mathsf{ws}(\xrightarrow{\text{ex}})$ *is equal to* $\bigcup_\ell (\mathbb{W}_\ell \times \mathbb{W}_\ell) \cap \xrightarrow{\text{ex}}$*, which also corresponds to the definition of* $\mathsf{ws}(X)$ *(see Sec. 3.2.2).*

- *Equality of read-from maps:*

  - $\mathsf{rf}(\xrightarrow{\text{ex}}) \subseteq \mathsf{rf}(X)$*: consider two events* x *and* y *such that* $(x,y) \in \mathsf{rf}(\xrightarrow{\text{ex}})$*. We want to show that* $(x,y) \in \mathsf{rf}(X)$*. By definition of* $\mathsf{rf}(X)$ *being well formed (see Sec. 3.2.1), we know there exists a unique* w_y *such that* $(w_y, y) \in \mathsf{rf}(X)$*. We want to show that* $x = w_y$*. Suppose as a contradiction that* $x \neq w_y$*, in this case we have either* $(x, w_y) \in \mathsf{ws}(X)$ *or* $(w_y, x) \in \mathsf{ws}(X)$*.*

    * *Suppose* $(x, w_y) \in \mathsf{ws}(X)$*. Since* $(w_y, y) \in \mathsf{rf}(X)$*, we know that* w_y *is a previous write to* y *in* $\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po}}$*. Indeed if* $(w_y, y) \in \mathsf{rfe}(X)$*, then* $(w_y, y) \in \mathsf{ghb}(X)$*, hence* $w_y \xrightarrow{\text{ex}} y$*. Otherwise* $w_y \xrightarrow{\text{po}} y$*. Since*

Function	Comment
$\mathrm{MM} \triangleq \lambda(E, X).\,((\mathbb{M} \times \mathbb{M}) \cap \mathsf{po}(X))$	two memory events in program order
$\mathrm{RM} \triangleq \lambda(E, X).\,((\mathbb{R} \times \mathbb{M}) \cap \mathsf{po}(X))$	a read followed by a memory event in program order
$\mathrm{WW} \triangleq \lambda(E, X).\,((\mathbb{W} \times \mathbb{W}) \cap \mathsf{po}(X))$	two writes in program order
$\mathrm{MW} \triangleq \lambda(E, X).\,((\mathbb{M} \times \mathbb{W}) \cap \mathsf{po}(X))$	a memory event followed by a write in program order

Figure 4.4: Notations to Extract Pairs From $\xrightarrow{\text{po}}$

$(x, w_y) \in \mathsf{ws}(X)$, we know $(x, w_y) \in \mathsf{ghb}(X)$, hence $x \xrightarrow{\text{ex}} w_y$. Hence w_y contradicts the maximality of x in the set of previous writes to y w.r.t. $\xrightarrow{\text{ex}}$.

- ∗ *Suppose $(w_y, x) \in \mathsf{ws}(X)$. In this case, we have $(y, x) \in \mathsf{fr}(X)$, hence in $\xrightarrow{\text{ex}}$ since $\xrightarrow{\text{fr}}$ is included in $\xrightarrow{\text{ghb}}$, which $\xrightarrow{\text{ex}}$ linearly extends. Moreover, we know that $(x, y) \in \mathsf{rf}(\xrightarrow{\text{ex}})$, hence $(x, y) \in \xrightarrow{\text{ex}}$. Thus, we have a cycle in $\xrightarrow{\text{ex}}$, a contradiction.*

– $\mathsf{rf}(X) \subseteq \mathsf{rf}(\xrightarrow{\text{ex}})$*: consider two events x and y such that $(x, y) \in \mathsf{rf}(X)$. We want to show that $(x, y) \in \mathsf{rf}(\xrightarrow{\text{ex}})$, i.e. that x is a previous write to y in $\xrightarrow{\text{ex}}$, and is maximal in this set w.r.t. $\xrightarrow{\text{ex}}$.*

- ∗ *Previous write: x is a write to y's location since they are in $\mathsf{rf}(X)$. Suppose by contradiction that $y \xrightarrow{\text{ex}} x$. In this case, we have $(y, x) \in \mathsf{fr}(X)$, hence a cycle in $\mathsf{rf}(X) \cup \mathsf{fr}(X)$, a contradiction.*
- ∗ *Maximality: As a contradiction, suppose that x is not a maximal previous write to y w.r.t. $\xrightarrow{\text{ex}}$. In this case, there exists a write w_y to y, such that $x \xrightarrow{\text{ex}} w_y \xrightarrow{\text{ex}} y$. Since $x \xrightarrow{\text{ex}} w_y$ and they are both writes to the same location, we have $(x, w_y) \in \mathsf{ws}(X)$. In this case, we have $(y, w_y) \in \mathsf{fr}(X)$. We also know that $w_y \xrightarrow{\text{ex}} y$. Since $\xrightarrow{\text{ex}}$ is a linear extension of $\mathsf{ghb}(X)$ in which is included $\mathsf{fr}(X)$, we have $y \xrightarrow{\text{ex}} w_y$. Hence we have a cycle in $\xrightarrow{\text{ex}}$, a contradiction.* □

4.2 A Hierarchy of Classical Models

We write $\mathsf{po}(E)$ (resp. $\mathsf{dp}(E, X)$, $\mathsf{rf}(X)$, $\mathsf{rfe}(X)$) for the function extracting the $\xrightarrow{\text{po}}$ (resp. $\xrightarrow{\text{dp}}$, $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{rfe}}$) relation from (E, X). We define notations to extract pairs of memory events from the program order in Fig. 4.4. For example, WW represents the function which extracts the write-write pairs in the program order of an execution. We write $\mathrm{W}_\ell\mathrm{W}_\ell$ when the writes have the same location ℓ.

We give in Fig. 4.5 a table summarising the implementation of these models in our framework. Note that all of these models consider the stores

Name	Arch	Sec.
SC	$(\mathrm{MM}, \mathsf{rf}, \lambda(E,X).\emptyset)$	4.2.1
TSO	$(\lambda(E,X).\,(\mathrm{RM}(E,X) \cup \mathrm{WW}(E,X))\,, \mathsf{rfe}, \lambda(E,X).\emptyset)$	4.2.2.2
PSO	$(\lambda(E,X)\,.\,\mathrm{RM}(E,X), \mathsf{rfe}, \lambda(E,X).\emptyset)$	4.2.2.3
RMO	$(\lambda(E,X)\,.\,\mathsf{dp}(E,X), \mathsf{rfe}, \lambda(E,X).\emptyset)$	4.2.2.4
Alpha	$(\lambda(E,X)\,.\,(\bigcup_\ell \mathrm{R}_\ell\mathrm{R}_\ell(E,X)), \mathsf{rfe}, \lambda(E,X).\emptyset)$	4.2.3

Figure 4.5: Summary of Models

to be atomic. We will see an example of model relaxing the store atomicity with the Power model presented in Chap. 8.

4.2.1 Sequential Consistency (SC)

SC has been defined in [Lam79] as follows:

> *[...] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*

4.2.1.1 Definition

SC allows no reordering of events ($\xrightarrow{\text{ppo}}$ equals $\xrightarrow{\text{po}}$ on memory events) and makes writes available to all processors as soon as they are issued ($\xrightarrow{\text{rf}}$ is global). Thus, there is no need for barriers, and any architecture definable in our framework is weaker than SC:

Definition 38 (Sequential Consistency)

$$SC \triangleq (\mathrm{MM}, \mathsf{rf}, \lambda(E,X).\emptyset)$$

4.2.1.2 Characterisation

The following criterion characterises, as in Sec. 3.5, valid SC executions on any architecture:

Definition 39 (SC Check)

$$A\,.\,\mathsf{check}_{SC}(E,X) = \mathsf{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}})$$

Indeed we show that, given an architecture A weaker than SC, any execution X valid on A satisfying this criterion is valid on SC:

Corollary 1 (SC Characterisation)

$$\forall A, (A \leq SC) \Rightarrow$$
$$(\forall EX, A\,.\,\mathsf{valid}(E,X) \wedge A\,.\,\mathsf{check}_{SC}(E,X) \Leftrightarrow SC\,.\,\mathsf{valid}(E,X))$$

Proof

$\Rightarrow$ *As* $\xrightarrow{\text{po}} \cup \xrightarrow{\text{com}} = SC\,.\,\mathsf{ghb}(E,X)$*, this is a direct consequence of Thm. 2.*

$\Leftarrow$ *As* $A \leq SC$*, this is a direct consequence of Thm. 1.* □

4.2.1.3 Equivalence With the Native One

In [Lam79], a SC execution is a total order $\xrightarrow{\text{ex}}$ which includes the program order:

$$SC\,.\,\mathsf{native}(\xrightarrow{\text{ex}}) \triangleq \mathsf{total\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge \xrightarrow{\text{po}} \subseteq \xrightarrow{\text{ex}}$$

The implicit execution model of [Lam79] states that a read r takes its value from the most recent write that precedes it in $\xrightarrow{\text{ex}}$. Hence we extract $\xrightarrow{\text{rf}}$ and $\xrightarrow{\text{ws}}$ from $\xrightarrow{\text{ex}}$ following Sec. 4.1, and build one of our execution witnesses from $\xrightarrow{\text{ex}}$:

$$SC\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) \triangleq (\mathbb{E}, \xrightarrow{\text{po}}, \mathsf{rf}(\xrightarrow{\text{ex}}), \mathsf{ws}(\xrightarrow{\text{ex}}))$$

Finally, we show, following the proof given in Sec. 4.1.2, that each execution witness built as above corresponds to a valid execution in our SC model:

Theorem 3 (SC Is SC)

$$\forall EX,\ SC\,.\,\mathsf{valid}(E,X) \Leftrightarrow \exists \xrightarrow{\text{ex}},\ SC\,.\,\mathsf{native}(\xrightarrow{\text{ex}}) \wedge SC\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

4.2.2 The Sparc Hierarchy

We present here the definitions of Sun's TSO, PSO and RMO. We will see in Sec. 10.3.3.2 how we indeed show that TSO can be obtained by PSO by barriers placement, and similarly PSO can be obtained from RMO, as specified in the Sparc documentation [spa94a, V9].

4.2.2.1 The *Value* Axiom

The execution model of the Sparc architectures is provided by the *Value* axiom of [spa94a], which states that a read (L_a for Sparc) reads from the most recent write (S_a) before L_a in the global ordering relation ($\leq$) or in the program order (;):

$$\mathit{Val}(L_a) = \mathit{Val}(\max_{\leq}\{S_a \mid S_a \leq L_a \vee S_a; L_a\})$$

The *Value* axiom of [spa94a, V8; App. K; p.283]

> *[...] states that the value of a data load is the value written by the most recent store to that location. Two terms combine to define the most recent store. The first corresponds to stores by other processors, while the second corresponds to stores by the processor that issued the load.*

This means that a $\xrightarrow{\text{rf}}$ relation occurs in the global order relation if and only if it is a $\xrightarrow{\text{rf}}$ between two events from distinct processors. Therefore, we deduce that for each of the Sparc architecture, the external $\xrightarrow{\text{rf}}$ are global, and the internal $\xrightarrow{\text{rf}}$ are not.

4.2.2.2 Total Store Order (TSO)

Definition TSO allows two relaxations [AG95]: *write to read program order*, and *read own write early.* The *write to read program order* relaxation means that TSO's preserved program order includes all pairs but the store-load ones. The *read own write early* relaxation means TSO's internal read-from maps are not global, which is also expressed by the *Value* axiom. We elide here the barriers semantics:

Definition 40 (*TSO^{ϵ}*)

$$\begin{aligned} ppo_{TSO} &\triangleq (\lambda(E,X).\,(\mathrm{RM}(E,X) \cup \mathrm{WW}(E,X)) \\ TSO^{\epsilon} &\triangleq (ppo_{TSO}, \mathsf{rfe}, \lambda(E,X).\emptyset) \end{aligned}$$

Characterisation Sec. 3.5 shows that the following criterion characterises valid executions (*w.r.t.* any $A \leq TSO$) that would be valid on TSO^{ϵ}, *e.g.* in Fig. 3.1:

Definition 41 (TSO Check)

$$A\,.\,\mathsf{check}_{TSO}(X) = \mathsf{acyclic}(\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{ppo-tso}})$$

Indeed we show that, given an architecture A weaker than TSO, any execution X valid on A satisfying this criterion is valid on TSO:

Corollary 2 (TSO Characterisation)

$$\begin{gathered} \forall A, (A \leq TSO) \Rightarrow \\ (\forall EX, A\,.\,\mathsf{valid}(E,X) \wedge A\,.\,\mathsf{check}_{TSO}(E,X) \Leftrightarrow TSO\,.\,\mathsf{valid}(E,X)) \end{gathered}$$

Proof

$\Rightarrow$ *As* $\xrightarrow{\text{ppo-tso}} \cup \xrightarrow{\text{hb-tso}} = TSO\,.\,\mathsf{ghb}(E,X)$*, this is a direct consequence of Thm. 2.*

$\Leftarrow$ *As* $A \leq TSO$*, this is a direct consequence of Thm. 1.* □

Equivalence With the Native One Sparc [spa94a, V. 8, Appendix K] formally defines a TSO execution axioms. We formulate[1] those axioms as follows:

$$\begin{array}{rcl}\mathsf{ptso}(\xrightarrow{\text{ex}}) & \triangleq & \mathsf{partial\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge \mathbb{RM} \subseteq \xrightarrow{\text{ex}} \wedge \mathbb{WW} \subseteq \xrightarrow{\text{ex}} \wedge \\ & & \exists \xrightarrow{\text{tso}}, \xrightarrow{\text{tso}} \subseteq \xrightarrow{\text{ex}} \wedge \mathsf{total\text{-}order}(\xrightarrow{\text{tso}}, \mathbb{W})\end{array}$$

Finally, we show, following the proof given in Sec. 4.1.2, that a partial order $\xrightarrow{\text{ex}}$ satisfying the axioms of Sun's TSO's specification corresponds to a valid execution in our TSO model:

Theorem 4 (TSO Is TSO)

$$\forall EX, \mathit{TSO}^{\epsilon} \,.\, \mathsf{valid}(E, X) \Leftrightarrow \exists \xrightarrow{\text{ex}}, \mathsf{ptso}(\xrightarrow{\text{ex}}) \wedge \mathit{TSO} \,.\, \mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

4.2.2.3 Partial Store Order (PSO)

Definition PSO maintains only the write-write pairs to the same location and all read-read and read-write pairs [spa94a]. However, there is no need to specify the write-write pairs to the same location in PSO's preserved program order. Indeed, according to Sec. 3.4.1.2, we know that two writes in program order to the same location are in $\xrightarrow{\text{ws}}$. We know, by hypothesis of our framework, that $\xrightarrow{\text{ws}}$ is always global. Hence, there is no need to specify write-write pairs to the same location in PSO's preserved program order, since we know that they are preserved globally (*i.e.* in $\xrightarrow{\text{ghb}}$) by the uniproc check. As the *Value* axiom holds for PSO as well, PSO's external $\xrightarrow{\text{rf}}$ are global whereas its internal $\xrightarrow{\text{rf}}$ are not:

Definition 42 (PSO^{ϵ})

$$\begin{array}{rcl}\mathit{ppo}_{PSO} & \triangleq & \lambda(E, X) \,.\, \mathrm{RM}(E, X) \\ \mathit{PSO}^{\epsilon} & \triangleq & (\mathit{ppo}_{PSO}, \mathsf{rfe}, \lambda(E, X) \,.\, \emptyset)\end{array}$$

Characterisation Sec. 3.5 shows that the following criterion characterises valid executions (*w.r.t.* any $A \leq PSO$) that would be valid on PSO^{ϵ}, *e.g.* in Fig. 3.1:

Definition 43 (PSO Check)

$$A \,.\, \mathsf{check}_{PSO}(E, X) = \mathsf{acyclic}(\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{ppo-pso}})$$

Indeed we show that, given an architecture A weaker than PSO, any execution (E, X) valid on A satisfying this criterion is valid on PSO:

[1] We omit the axioms *Atomicity* and *Termination.*

Corollary 3 (PSO Characterisation)

$$\forall A, (A \leq PSO) \Rightarrow$$
$$(\forall EX, A\,.\,\mathsf{valid}(E,X) \wedge A\,.\,\mathsf{check}_{PSO}(E,X) \Leftrightarrow PSO\,.\,\mathsf{valid}(E,X))$$

Proof

$\Rightarrow$ *As* $\xrightarrow{\text{ppo-tso}} \cup \xrightarrow{\text{hb-pso}} = PSO\,.\,\mathsf{ghb}(E,X)$*, this is a direct consequence of Thm. 2.*

$\Leftarrow$ *As* $A \leq PSO$*, this is a direct consequence of Thm. 1.* □

Equivalence With the Native One Sparc [spa94a, V. 8, Appendix K] formally defines a PSO execution as a partial order $\xrightarrow{\text{ex}}$ on memory events constrained by some axioms. We formulate those as follows:

$$\mathsf{ppso}(\xrightarrow{\text{ex}}) \triangleq \mathsf{partial\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge \left(\mathbb{RM} \cup \bigcup_{\ell} \mathbb{W}_{\ell}\mathbb{M}_{\ell}\right) \subseteq \xrightarrow{\text{ex}}$$

We define Sun PSO's definition of a valid execution as follows, since the *Value* axiom holds for PSO as well:

$$PSO\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) \triangleq (\mathbb{E}, \xrightarrow{\text{po}}, \mathsf{rf}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po}}), \mathsf{ws}(\xrightarrow{\text{ex}}))$$

We show, following the proof given in Sec. 4.1.2, that any partial order $\xrightarrow{\text{ex}}$ satisfying Sun PSO's specification corresponds to a valid execution of our PSO model:

Theorem 5 (PSO Is PSO)

$$\forall EX, PSO^{\epsilon}\,.\,\mathsf{valid}(E,X) \Leftrightarrow \exists \xrightarrow{\text{ex}}, \mathsf{ppso}(\xrightarrow{\text{ex}}) \wedge PSO\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

4.2.2.4 Relaxed Memory Order (RMO)

Definition RMO preserves the program order between the write-write and read-write pairs to the same location, and the read-read and read-write pairs in dependency [spa94a]. However, there is no need to specify the write-write and read-write pairs to the same location in RMO's preserved program order, as exposed in Sec. 3.4.1.2. Indeed, the write-write pairs to the same location are in $\xrightarrow{\text{ws}}$, hence in $\xrightarrow{\text{ghb}}$. Moreover, by the same reasoning, the read-write pairs to the same location are in $\xrightarrow{\text{fr}}$, hence in $\xrightarrow{\text{ghb}}$.

The *Value* axiom holds for RMO as well. Hence we have (writing $\mathsf{dp}(E,X)$ for the pairs in dependency in an execution (E,X)):

Definition 44 (RMO^{ϵ})

$$ppo_{RMO} \triangleq \lambda(E,X)\,.\,\mathsf{dp}(E,X)$$
$$RMO^{\epsilon} \triangleq (ppo_{RMO}, \mathsf{rfe}, \lambda(E,X)\,.\,\emptyset)$$

Characterisation Sec. 3.5 shows that the following criterion characterises valid executions (*w.r.t.* any $A \leq RMO$) that would be valid on RMO^{ϵ}, *e.g.* in Fig. 3.1:

Definition 45 (RMO Check)

$$A\,.\,\mathsf{check}_{\mathsf{RMO}}(E, X) = \mathsf{acyclic}(\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{ppo-rmo}})$$

Indeed we show that, given an architecture A weaker than RMO, any execution (E, X) valid on A satisfying this criterion is valid on RMO:

Corollary 4 (RMO Characterisation)

$$\forall A, (A \leq RMO) \Rightarrow$$
$$(\forall EX, A\,.\,\mathsf{valid}(E, X) \wedge A\,.\,\mathsf{check}_{\mathsf{RMO}}(E, X) \Leftrightarrow RMO\,.\,\mathsf{valid}(E, X))$$

Proof

$\Rightarrow$ *As* $\xrightarrow{\text{ppo-rmo}} \cup \xrightarrow{\text{hb-rmo}} = RMO\,.\,\mathsf{ghb}(E, X)$, *this is a consequence of Thm. 2.*

$\Leftarrow$ *As* $A \leq RMO$, *this is a consequence of Thm. 1.* □

Equivalence With the Native One Sparc [spa94a, V. 8, Appendix K] formally defines a RMO execution as a partial order $\xrightarrow{\text{ex}}$ on memory events constrained by some axioms:

$$\mathsf{prmo}(\xrightarrow{\text{ex}}) \quad \triangleq \quad \mathsf{partial\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge (\xrightarrow{\text{dp}} \cup \bigcup_{\ell} \mathbb{M}_{\ell}\mathbb{W}_{\ell}) \subseteq \xrightarrow{\text{ex}}$$

We define Sun RMO's definition of a valid execution as follows, since the *Value* axiom holds for RMO as well:

$$\text{RMO}\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) \quad \triangleq \quad (\mathbb{E}, \xrightarrow{\text{po}}, \mathsf{rf}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po}}), \mathsf{ws}(\xrightarrow{\text{ex}}))$$

We show, following the proof given in Sec. 4.1.2, that any partial order $\xrightarrow{\text{ex}}$ satisfying Sun RMO's specification corresponds to a valid execution of our RMO model:

Theorem 6 (RMO Is RMO)

$$\forall EX, RMO^{\epsilon}\,.\,\mathsf{valid}(E, X) \Leftrightarrow \exists \xrightarrow{\text{ex}}, \mathsf{prmo}(\xrightarrow{\text{ex}}) \wedge RMO\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

4.2.3 Alpha

4.2.3.1 Definition

Alpha maintains the write-write, read-read and read-write pairs to the same location [alp02]. We exposed in Sec. 3.4.1.2 why there is no need to include the write-write pairs to the same location in $\xrightarrow{\text{ppo}}$ (they are in $\xrightarrow{\text{ws}}$ thus in $\xrightarrow{\text{ghb}}$ by uniproc), and why the read-write pairs to the same location are exempted as well (they are in $\xrightarrow{\text{fr}}$ thus in $\xrightarrow{\text{ghb}}$ by uniproc). However, we do need to specify the read-read pairs to the same location in Alpha's preserved program order, because such a relation may not be global, if not specified in the $\xrightarrow{\text{ppo}}$. Hence we have. Moreover, Alpha specifies, for every read, the write from which it reads as the last one either:

- in *processor issue sequence*, *i.e.* our program order, or
- in the *BEFORE* order, which corresponds to our global happens-before order.

Thus, similarly to the Sun models, external $\xrightarrow{\text{rf}}$ are global whereas internal are not:

Definition 46 (*Alpha*$^\epsilon$)

$$\begin{array}{rcl} \mathit{ppo}_{\mathit{Alpha}} & \triangleq & \lambda(E,X)\,.\,(\bigcup_\ell \mathrm{R}_\ell\mathrm{R}_\ell(E,X)) \\ \mathit{Alpha}^\epsilon & \triangleq & (\mathit{ppo}_{\mathit{Alpha}}, \mathsf{rfe}, \lambda(E,X)\,.\,\emptyset) \end{array}$$

4.2.3.2 Characterisation

Sec. 3.5 shows the following criterion characterises valid executions (*w.r.t.* any $A \leq \mathit{Alpha}$) that would be valid on Alpha^ϵ, *e.g.* in Fig. 3.1:

Definition 47 (Alpha Check)

$$A\,.\,\mathsf{check}_{\mathit{Alpha}}(E,X) = \mathsf{acyclic}(\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rfe}} \cup \xrightarrow{\text{ppo-alpha}})$$

Indeed we show that this criterion characterises valid executions that are valid *w.r.t.* Alpha^ϵ on any $A \leq \mathit{Alpha}$:

Corollary 5 (Alpha Characterisation)

$$\begin{array}{c} \forall A, (A \leq \mathit{Alpha}) \Rightarrow \\ (\forall EX, A\,.\,\mathsf{valid}(E,X) \wedge A\,.\,\mathsf{check}_{\mathit{Alpha}}(E,X) \Leftrightarrow \mathit{Alpha}\,.\,\mathsf{valid}(E,X)) \end{array}$$

Proof

$\Rightarrow$ *This is a direct consequence of Thm. 2.*

$\Leftarrow$ *As $A \leq \mathit{Alpha}$, this is a direct consequence of Thm. 1.* □

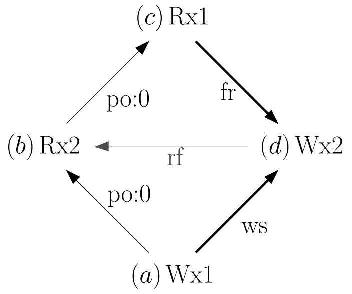
4.2.3.3 Equivalence With the Native One

Alpha [alp02] formally defines an Alpha execution as a partial order $\xrightarrow{\text{ex}}$ on memory events constrained by some axioms. We formulate those axioms as follows:

$$\begin{array}{rcl} \mathsf{palpha}(\xrightarrow{\text{ex}}) & \triangleq & \mathsf{partial\text{-}order}(\xrightarrow{\text{ex}}, \mathbb{E}) \wedge (\bigcup_\ell \mathbb{R}_\ell\mathbb{M}_\ell \cup \bigcup_\ell \mathbb{W}_\ell\mathbb{W}_\ell) \subseteq \xrightarrow{\text{ex}} \wedge \\ & & \mathsf{acyclic}(\mathsf{rf}(\xrightarrow{\text{ex}}) \cup \mathsf{dp}(\xrightarrow{\text{ex}})) \end{array}$$

P_0	P_1
(a) `x ← 1`	(d) `x ← 2`
(b) `r1 ← x`	
(c) `r2 ← x`	
Observed? `r1=2; r2=1; x=2`	

(a) Program



(b) An Execution Witness

Figure 4.6: Load-Load Hasard Example

Since Alpha's definition of read-from map corresponds to the Sparc one, we extract the $\xrightarrow{\text{rf}}$ from an Alpha execution as we did for the Sparc hierarchy, and define a valid Alpha execution as follows:

$$\mathit{Alpha}\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) \quad \triangleq \quad (\mathbb{E}, \xrightarrow{\text{po}}, \mathsf{rf}(\xrightarrow{\text{ex}} \cup \xrightarrow{\text{po}}), \mathsf{ws}(\xrightarrow{\text{ex}}))$$

Finally, we show, following the proof given in Sec. 4.1.2, that any partial order $\xrightarrow{\text{ex}}$ satisfying Alpha's axioms corresponds to a valid execution in our Alpha model:

Theorem 7 (Alpha Is Alpha)

$$\forall EX, \mathit{Alpha}^{\epsilon}\,.\,\mathsf{valid}(E, X) \Leftrightarrow \exists \xrightarrow{\text{ex}}, \mathsf{palpha}(\xrightarrow{\text{ex}}) \wedge \mathit{Alpha}\,.\,\mathsf{wit}(\xrightarrow{\text{ex}}) = X$$

4.2.4 RMO and Alpha Are Incomparable

4.2.4.1 Load-Load Hasard

We exposed in Sec. 3.4.1.3 that RMO authorises load-load hasard, where two reads on the same processor from the same location can be reordered. As an illustration, we explained why the example of Fig. 3.10(a) can exhibit its outcome on a RMO machine. We recall this example in Fig. 4.6.

However, Alpha includes read-read pairs to the same location in its preserved program order, as exposed in Sec. 4.2.3. Therefore the read-read pair $(b) \xrightarrow{\text{po}} (c)$ to the same location on P_0 is included in Alpha's preserved program order. Moreover, we know that the external read-from maps are global on Alpha, hence the $\xrightarrow{\text{rf}}$ relation $(d) \xrightarrow{\text{rf}} (b)$ is global as well. Hence the execution (E, X) depicted in Fig. 3.10(b) exhibits a cycle in $\mathit{Alpha}\,.\,\mathsf{ghb}(E, X)$: $(b) \xrightarrow{\text{ppo}} (c) \xrightarrow{\text{fr}} (d) \xrightarrow{\text{rfe}} (b)$, which forbids this execution.

Hence, RMO authorises load-load hasard whereas Alpha does not.

iriw

P_0	P_1	P_2	P_3
(a) `r1` ← `x` (b) `r2` ← `y`	(c) `r2` ← `y` (d) `r1` ← `x`	(e) `x` ← `1`	(f) `y` ← `2`

Observed? `0:r1=1`; `0:r2=0`; `1:r2=2`; `1:r1=0`;

Figure 4.7: The **iriw** Example

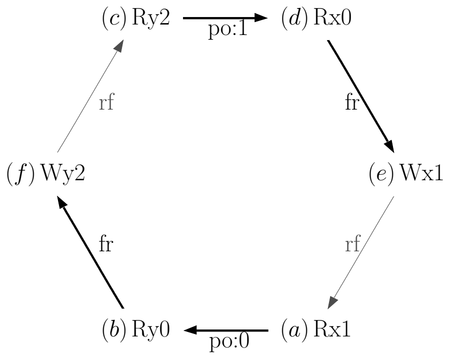


Figure 4.8: An non-SC execution of **iriw**

4.2.4.2 Iriw With Dependencies

Consider now the **iriw** (for Independent Reads of Independent Writes) example given in Fig. 4.7, and suppose that there is a dependency between the pairs of reads on P_0 and P_1, *i.e.* $(a) \xrightarrow{\text{dp}} (b)$ and $(c) \xrightarrow{\text{dp}} (d)$. We can enforce these pairs to be in dependency by adding for example a logical operation between them, such as a `xor` operating on the registers of the load instructions associated to the read events.

The specified outcome may be revealed by an execution such as the one we depict in Fig. 4.8. Suppose that each location and register initially hold 0. If `r1` holds 1 on P_0 in the end, the read (a) has read its value from the write (e) on P_2, hence $(e) \xrightarrow{\text{rf}} (a)$. On the contrary, if `r2` holds 0 in the end, the read (b) has read its value from the initial state, thus before the write (f) on P_3, hence $(b) \xrightarrow{\text{fr}} (f)$. Similarly, we have $(f) \xrightarrow{\text{rf}} (c)$ from `r2` holding 1 on P_1, and $(d) \xrightarrow{\text{fr}} (e)$ from `r1` holding 0 on P_1. Hence, if the specified outcome is observed, it ensures that at least one execution of the test contains the cycle depicted in Fig. 4.8:

$$(a) \xrightarrow{\text{dp}} (b) \xrightarrow{\text{fr}} (f) \xrightarrow{\text{rfe}} (c) \xrightarrow{\text{dp}} (d) \xrightarrow{\text{fr}} (e) \xrightarrow{\text{rf}} (a)$$

This cycle is not global on Alpha whereas it is on RMO. This means that the associated execution is authorised on Alpha whereas it is forbidden on RMO. Indeed on RMO, the pairs in dependency are included in the preserved program order, hence the pairs $(a) \xrightarrow{\text{dp}} (b)$ and $(c) \xrightarrow{\text{dp}} (d)$ are included in $\xrightarrow{\text{ppo}}$, hence in $\xrightarrow{\text{ghb}}$. Moreover, the external read-from maps are global on RMO (see Sec. 4.2.2.4), and $\xrightarrow{\text{fr}}$ is always global. However on Alpha, these pairs are not preserved globally, therefore this execution is authorised.

Hence, Alpha authorises **iriw** with dependencies whereas RMO does not.

Chapter 5

Related Work

We present here three kinds of related work. In Sec. 5.1 we present several generic weak memory models. In Sec. 5.2, we examine related work according to the view of memory they use, either global-time like in our framework, or using a per-processor *view order*. Finally, in Sec. 5.3 and Sec. 5.5, we present related work according to their style, either axiomatic like us, *operational*, or specifications of weak memory models as *program transformations*.

5.1 Generic Models

The work that is the closest to ours is probably W. Collier's [Col92]. He presents several abstract models in terms of the relaxations (*w.r.t.* SC) they allow. However he does not address the store atomicity relaxation.

S. Adve and K. Gharachorloo's tutorial gives a categorisation of memory models, in which they give intuition about the relaxations in terms of the actual hardware, *i.e.* store buffers and cache lines. By contrast, we chose to abstract from the implementation.

S. Adve [Adv93] and K. Gharachorloo [Gha95] both present in their thesis a generic framework. S. Adve's work focuses on the notion of *data race freeness*, and defines and studies models which enforce the *data race freeness guarantee*. We choose to examine this property on top of our framework, and see which conditions enforce this guarantee for its instances, instead of building a model with a hard-wired data race free guarantee. K. Gharachorloo's work focuses on the implementation and performance of several weak memory models. We choose to give an abstract view of the memory, because we want to provide a model in which the programmer does not have to care about the minute details of the implementation—which are often secret—to write correct programs.

Finally, the Nemos [YGLS04] framework covers a broad range of models including Itanium as the most substantial example. Itanium [ita02] is rather different from the models we presented, or from the Power model we present in Chap. 8; we do not know whether our framework could handle such a model or whether a satisfactory Power model could be expressed in Nemos. Indeed, Itanium uses several events per instruction, whereas we represent instructions by only one memory event. Moreover, Itanium's model specifies the semantics not only of stores, loads and fences, but also of *load-acquire* and *store-release* instructions. By contrast, we chose to specify the semantics of more atomic constructions, and build the semantics of derived constructions on top of them, as we will see in Chap. 10.

5.2 Global-Time vs. View Orders

We can distinguish weak memory models according to the view of memory they present. Such models are either in terms of a global time line in which the memory events are embedded, or provide one view order per processor.

Most of the documentations that provide a formal model, *e.g.* Alpha [alp02] and Sun [spa94a], are in terms of a global time line. We believe this provides a usable model to the programmer, because it abstracts from the implementation's details. Moreover such a model allows the vendor to provide a formal and usable model without revealing the secrets of the implementation.

Some memory models are in terms of view orders, *e.g.* [AAS03] and the Power documentation [pow09]. A. Adir *et al.* 's work focuses on the PowerPC model, and presents numerous axioms describing a pre-cumulativity (pre Power 4) version of Power. We find such models difficult to understand, because they provide several axioms, often without clear intuition.

5.3 Axiomatic vs. Operational

Formal models roughly fall into two classes: operational models and axiomatic models. Operational models, e.g. [BPa, YGL, HKV98], are abstractions of actual machines composed of idealised hardware components such as queues. They seem appealingly intuitive and offer a relatively direct path to simulation, at least in principle. However, they require a precise knowledge of the implementation of the hardware.

Axiomatic models focus on the segregation of allowed and forbidden behaviours, usually by constraining various order relations on memory accesses; they are well adapted for model exploration, as we do in Chap. 7. Several of the most formal vendor specifications have been in this style [alp02, spa94a, ita02].

5.4 Characterisation of Behaviours

The characterisation we propose in Thm. 2 is simple, since it is merely an acyclicity check of the global happens-before relation. This check is already known for *SC* [LHH91], and recent verification tools use it for architectures with store buffer relaxation [HVM$^+$, BMa]. We believe our work could help extending these methodologies to models relaxing store atomicity.

5.5 Memory Models As Program Transformations

Another style of weak memory models' specification has recently emerged, *e.g.* in S. Burckhardt *et al.* 's work [BMS] or R. Ferreira *et al.* 's [FFS]. This line of research specifies weak memory models as program transformations. Instead of specifiying a transition system as in an operational style, rewriting rules apply to the program as a whole, to represent the effect of the memory model on this program's behaviour. Although this approach addresses only a limited store atomicity relaxation, S. Burckhardt *et al.* 's work has an elegant flavour of testing semantics and realisability, as we shall develop in Chap. 14.

Part III

Testing Weak Memory Models

A program running on a multiprocessor behaves *w.r.t.* the *memory model* of the architecture. But, as we know, some public documentations [int07, pow09] lack formal definitions of these models. Hence we rely on intensive testing, *via litmus* tests, to design and study such models. A litmus test is a small program in pseudo or assembly code, specifying an initial state and a condition on the final state. Running such a test on a given machine and checking whether the final condition is verified or not may reveal *relaxations* of the architecture of the machine.

We first give an overview of the relaxations we studied in the Power architecture, and their syntax. Then, we present in Chap. 7 the tool we wrote to systematise and automatise the production of litmus tests, in Power or x86 assembly. We present in Chap. 8 the way we used this tool and applied our testing methodology to produce a model for the Power architecture [pow09], which we also detail in the same chapter.

Chapter 6

Relaxations

We want to do *model exploration*: for a given machine—or set of machines—agreeing with a certain architecture, we want to be able, by testing, to highlight the parameters (*w.r.t.* the framework presented in Chap. 3) that this machine displays. Thus, we need to be able to test whether the relations $\xrightarrow{\text{ppo}}$ or $\xrightarrow{\text{rf}}$ are global or not on this machine. More precisely, we want to determine which subrelation of the program order $\xrightarrow{\text{po}}$ is global: this will form the preserved program order $\xrightarrow{\text{ppo}}$ of the machine we test. Similarly, we want to determine which subrelation of the read-from map $\xrightarrow{\text{rf}}$ is global.

We call any sequence of the relations defined in Chap. 3 a *candidate relaxation*. We detail below the syntax of the candidate relaxations we use. We consider a given candidate relaxation to be *relaxed*, or *non-global*, when we observe it to be exhibited on a machine. On the contrary, *safe* or *global* candidate relaxations are the candidate relaxations guaranteed, *e.g.* by the documentation, never to be exhibited. In our framework, global candidate relaxations are included in $\xrightarrow{\text{ghb}}$, and non-global ones are not.

6.1 A Brief Glance at the Power Documentation

The Power documentation [pow09] states several particularities. We detail them in the following section by presenting excerpts from the documentation. The words added by myself for clarity are between brackets, *e.g.* [Led Zeppelin]. The words erased for brevity or irrelevance are represented as dots between brackets, *i.e.* [. . .]. In particular, as we focus on giving a Power model for basic user mode, we do not consider any situation where the memory is Guarded, or Caching Inhibited [pow09, p. 657].

6.1.1 Axioms of Our Model

We highlight here some excerpts of the documentation that can be interpreted as a confirmation of some of the hypotheses of our model (see

Chap. 3).

6.1.1.1 Coherence

The memory is coherent [pow09, p. 657, 1st col, last §]:

> *Memory coherence refers to the ordering of stores to a single location. Atomic stores to a given location are* coherent *if they are serialized in some order, and no processor or mechanism is able to observe any subset of those stores as occurring in a conflicting order.*

Hence, we conclude that the write serialisation as described in Sec. 3.2.2 actually exists, and is indeed global, as we supposed in Sec. 3.3.1.

6.1.1.2 Uniprocessor

The *sequential execution model* is defined as follows [pow09, p. 29, 1st col, last §]:

> *The model of program execution in which the processor appears to execute one instruction at a time, completing each instruction before beginning to execute the next instruction is called the "sequential execution model".*

Our uniproc axiom may be related to this excerpt [pow09, p. 661, 2nd col, penultimate §]:

> *Because the storage model is weakly consistent, the sequential execution model as applied to instructions that cause storage accesses guarantees only that those accesses appear to be performed in program order with respect to the processor executing the instructions.*

We believe that this excerpt means that the sequential execution model actually ensures a guarantee similar to uniproc, and nothing more: in particular, one should not consider the uniproc check as a global check, since the excerpt explicitely states that this guarantee only holds for the particular processor that issued the accesses.

6.1.2 Store Buffering

Power allows store buffering [pow09, p. 661, 1st col, 1st bullet]:

> *If a Load instruction specifies the same storage location as a preceding Store instruction [. . .], the load may be satisfied from a "store queue" (a buffer into which the processor places stored values before presenting them to the storage sub-system), and not be visible to other processors and mechanisms.*

From this we can deduce that internal read-from maps are not global in Power. Since there is no mention of the external ones, we suppose that they are relaxed, which our experiment confirmed (see Sec. 8.1).

6.1.3 Load-Load Pairs

6.1.3.1 Data Dependencies

The load-load pairs related by a data dependency are preserved in the program order [pow09, p. 660, 1st col, last bullet]:

> *If a Load instruction depends on the value returned by a preceding Load instruction (because the value is used to compute the effective address specified by the second Load), the corresponding storage accesses are performed in program order with respect to any processor or mechanism [. . .]. This applies even if the dependency has no effect on program logic (e.g., the value returned by the first Load is ANDed with zero and then added to the effective address specified by the second Load).*

We consider indeed that the dependency over values described in this excerpt is a data dependency, as described in Sec. 6.2.4.1.

6.1.3.2 Semantics of the `isync` Barrier

The `isync` barrier, in conjunction with a branch instruction such as `bne`, forms a load-load barrier [pow09, p. 661, 1st col, 2nd §]:

> *[. . .] if an isync follows a conditional Branch instruction that depends on the value returned by a preceding Load instruction, the load on which the Branch depends is performed before any loads caused by instructions following the isync. This applies even if the effects of the "dependency" are independent of the value loaded (e.g., the value is compared to itself and the Branch tests the EQ bit in the selected CR field), and even if the branch target is the sequentially next instruction.*

Since `isync` is not mentioned to be cumulative, we consider it non-cumulative.

6.1.4 Load-Store Pairs

6.1.4.1 Data Dependencies

The load-store pairs related by a data dependency are preserved in the program order [pow09, p. 661, 1st col, 1st §]:

> *[...] if a Store instruction depends on the value returned by a preceding Load instruction (because the value returned by the Load is used to compute either the effective address specified by the Store or the value to be stored), the corresponding storage accesses are performed in program order.*

We consider indeed that the dependency over values described in this excerpt is a data dependency, as described in Sec. 6.2.4.1.

6.1.4.2 Control Dependencies

The load-store pairs related by a control dependency are preserved in the program order [pow09, p. 661, 1st col, 1st §]:

> *The same [as above, Sec. 6.1.4.1] applies if* whether *the Store instruction is executed depends on a conditional Branch instruction that in turn depends on the value returned by a preceding Load instruction.*

We consider indeed that the dependency over branches described in this excerpt is a control dependency, as described in Sec. 6.2.4.2.

6.1.5 Barriers

6.1.5.1 Ordering Induced by a Barrier

Plain Ordering is defined as follows [pow09, p. 660, 2nd col, 1st bullet]:

> *When a processor (P1) executes a* Synchronize *[*`sync`*] [...] instruction a* memory barrier *is created, which orders applicable storage accesses pairwise, as follows. Let A be a set of storage accesses that includes all storage accesses associated with instructions preceding the barrier-creating instruction, and let B be a set of storage accesses that includes all storage accesses associated with instructions following the barrier-creating instruction. For each applicable pair a_ib_j of storage accesses such that a_i is in A and b_j is in B, the memory barrier ensures that a_i will be performed with respect to any processor or mechanism [...], before b_j is performed with respect to that processor or mechanism.*

We consider here that "preceding" is *w.r.t.* program order. Hence a barrier orders certain pairs of accesses (the applicable ones): if two instructions i_1 and i_2 on the same processor are separated by a barrier in program order, then the associated events m_1 and m_2, if they form an applicable pair, will be globally ordered according to the program order.

Cumulativity is defined as follows [pow09, p. 660, 2nd col, penultimate §]:

> *The ordering done by a memory barrier is said to be "cumulative" if it also orders storage accesses that are performed by processors and mechanisms other than P1, as follows.*
>
> - *A includes all applicable storage accesses by any such processor or mechanism that have been performed with respect to P1 before the memory barrier is created.*
> - *B includes all applicable storage accesses by any such processor or mechanism that are performed after a [sentinel] Load instruction executed by that processor or mechanism has returned the value stored by a store that is in B.*

We took some liberty with this definition, as we shall see in Sec. 8.2.2.3. In particular, we did not formalise the notion of being "performed with respect to", either a processor or all processors. We did not take into account the notion of *sentinel load* for the B group either.

Yet, we took this excerpt as a source of inspiration for our definition of cumulativity. For group A, we consider the events that are in $\xrightarrow{\text{rf}}$ with an event preceding the barrier in program order. For group B, we consider the events that are in $\xrightarrow{\text{rf}}$ with an event following the barrier in program order. Thus, we consider a barrier to be *A-cumulative* when it imposes a global ordering to the chains $w \xrightarrow{\text{rf}} m_1 \xrightarrow{\text{po}} m_2$, provided that the barrier separates the instructions associated to m_1 and m_2 in program order. Similarly, we consider a barrier to be *B-cumulative* when it imposes a global ordering to the chains $m_1 \xrightarrow{\text{po}} m_2 \xrightarrow{\text{rf}} r$, provided the barrier separates the instructions associated to m_1 and m_2 in program order. We formalise these notions in Sec. 10.3.3.

6.1.5.2 The `sync` barrier

The `sync` barrier preserves all pairs in the program order [pow09, p. 700, 1st col, L=0 case]:

> *The memory barrier [*`sync`*] provides an ordering function for the storage accesses associated with all instructions that are executed by the processor executing the* `sync` *instruction. The applicable pairs are all pairs a_ib_j in which b_j is a data access [...].*

Or again, more clearly [pow09, p. 700, 2nd col, 1st bullet]:

> *Executing the [*`sync`*] instruction ensures that all instructions preceding the* `sync` *instruction have completed before the* `sync` *instruction completes, and that no subsequent instructions are initiated until after the* `sync` *instruction completes.*

Hence, we consider that if two instructions i_1 and i_2 are separated by a `sync` in program order, this imposes a global ordering between the associated events m_1 and m_2 according to the program order, regardless of the directions of m_1 and m_2.

Moreover, the `sync` barrier is cumulative [pow09, p.700, 1st col, last line]:

> *The ordering done by the memory barrier [*`sync`*] is cumulative*

That is, more precisely [pow09, p. 700, 2nd col, 3rd bullet]:

> *The memory barrier [*`sync`*] provides the additional ordering function such that if a given instruction that is the result of a store in set B is executed, all applicable storage accesses in set A have been performed with respect to the processor executing the instruction to the extent required by the associated memory coherence properties. The cumulative properties of the barrier apply to the execution of the given instruction as they would to a load that returned a value that was the result of a store in set B.*

We drift away from this definition and prefer to consider that `sync` is A- and B-cumulative in the sense we expose in Sec. 6.1.5.1, regardless of any direction of the events involved. We will see in Chap. 8 that our intensive testing did not contradict our liberal definition.

6.1.5.3 The `lwsync` barrier

The `lwsync` barrier preserves all pairs in the program order, except the store-load ones [pow09, p. 700, 1st col, L=1 case]:

> *The memory barrier [*`lwsync`*] provides an ordering function for the storage accesses caused by [...] instructions that are executed by the processor executing the* `[lw]sync` *instruction [...]. The applicable pairs are all pairs a_ib_j of such accesses except those in which a_i is an access caused by as Store [...] and b_j is an access caused by a Load instruction.*

Hence, we consider that if two instructions i_1 and i_2 are separated by a `lwsync` in program order, this imposes a global ordering between the associated events m_1 and m_2 according to the program order, provided that (m_1,m_2) does not belong to $\mathbb{W} \times \mathbb{R}$.

Moreover, the `lwsync` barrier is cumulative, except from stores to loads:

> *The ordering done by the memory barrier [*`lwsync`*] is cumulative.*

Once again, we drift away from this definition and prefer to consider that `lwsync` is A- and B-cumulative in the sense we expose in Sec. 6.1.5.1, provided that (e_1,e_2) does not belong to $\mathbb{W} \times \mathbb{R}$, where e_1 is the beginning of the chain $\xrightarrow{\text{rf}};\xrightarrow{\text{po}}$ (resp. $\xrightarrow{\text{po}};\xrightarrow{\text{rf}}$) and e_2 its end.

Candidate Relaxation	Comment
Rfi	a write followed by a read from the same location in program order
Rfe	a write and a read from the same location on distinct processors
Wsi	a write followed by a write to the same location in program order
Wse	two writes to the same location on distinct processors
Fri	a read followed by a write to the same location in program order
Fre	a read and a write to the same location on distinct processors

Figure 6.1: Table of Communication Candidate Relaxations

6.2 Candidate Relaxations

We present here the syntax of candidate relaxations, and how we implement a given candidate relaxation, in the light of the excerpts we presented above.

6.2.1 Communication Candidate Relaxations

We call communication candidate relaxations the candidate relaxations associated to the subrelations of $\xrightarrow{\text{com}}$, $i.e.$ $\xrightarrow{\text{rf}}$, $\xrightarrow{\text{fr}}$ and $\xrightarrow{\text{ws}}$. We give a table of their syntax in Fig. 6.1.

We implement an internal communication candidate relaxation ($i.e.$ Rfi, Wsi, Fri) with two instructions to the same location in the program order, with the appropriate directions. For example, Rfi corresponds to a write followed in $\xrightarrow{\text{po}}$ by a read to the same location, hence is implemented by a store followed by a load on the same processor and to the same location.

We implement an external communication candidate relaxation ($i.e.$ Rfe, Wse, Fre) with two instructions to the same location on distinct processors, with the appropriate direction. For example, Fre corresponds to a read preceding (in $\xrightarrow{\text{ghb}}$) a write to the same location but from a distinct processor, hence is implemented by a load and a store to the same location but on distinct processors.

6.2.2 Program Order Candidate Relaxations

We call program order candidate relaxations each relation between two events in the program order. Program order candidate relaxations have the following syntax:

$$Po(s \mid d)(R \mid W)(R \mid W)$$

Candidate Relaxation	Comment
PosRR	two reads in program order, to the same location
PodRR	two reads in program order, to distinct locations
PosRW	a read followed by a write to the same location in program order
PodRW	a read followed by a write to a different location in program order
PosWW	two writes to the same location in program order
PodWW	two writes to distinct locations in program order
PosWR	a write followed by a read from the same location in program order
PodWR	a write followed by a read from a different location in program order

Figure 6.2: Table Of Program Order Candidate Relaxations

where:

- s (resp. d) indicates that the two events are relative to the same (resp. different) location(s);
- R (resp. W) indicates an event to be a read (resp. a write).

We give in Fig. 6.2 a table summarising their syntax. By uniproc, we know that PosWW and PosRW correspond to Wsi and Fri respectively.

We implement a program order candidate relaxation by generating two memory instructions following one another on the same processor, according to the specified directions (store for W and load for R), with the same location if s is specified and distinct locations if d is specified.

6.2.3 Barriers Candidate Relaxations

We specify the presence of a fence instruction with Fenced candidate relaxations, similar to Po candidate relaxations, except that a fence instruction is inserted. We have the following syntax for these candidate relaxations:

$$\mathit{Fenced}(s \mid d)(R \mid W)(R \mid W)$$

The inserted fence is by default the stronger fence provided by the architecture, *i.e.* `mfence` for x86 and `sync` for Power.

Barriers can also be specified by using specific names. More precisely, we have MFence for x86, while on Power we have Sync and LwSync. For example, in order to yield two reads (RR) to different locations (d) and separated by the lightweight Power barrier `lwsync`, we specify LwSyncdRR. This may lead *e.g.* to the following code:

```
(1) lwz r1,0(r2)
(2) lwsync
(3) lwz r3,0(r4)
```

The instructions `lwz r1,0(r2)` at line (1) and `lwz r3,0(r4)` at line (3) are reads from distinct locations, separated by a `lwsync` barrier, at line (2).

6.2.4 Dependencies Candidate Relaxations

We distinguish the dependencies candidate relaxations between data and control dependencies.

6.2.4.1 Data Dependencies

These candidate relaxations have the following syntax:

$$Dp(s \mid d)(R \mid W)$$

where:

- s (resp. d) indicates that the two events are relative to the same (resp. different) location(s);
- R (resp. W) indicates that the target event is a read (resp. a write);

We do not need to specify the direction of the source event, since it is always a read by definition (see Sec. 3.4.1.3).

We implement a data dependency with a false dependency that operates on the address of the target memory access. See for instance the following code:

```
(1) lwz r1,0(r2)
(2) xor r3,r1,r1
(3) stwx r4,r3,r5
```

The instructions `lwz r1,0(r2)` at line (1) and `stwx r4,r3,r5` at line (3) correspond to a DpdW candidate relaxation. They are indeed in data dependency, as they are separated by `xor r3,r1,r1` at line (2). Hence the address of the indexed store `stwx r4,r3,r5` depends on the content of the index register `r3`, which itself depends on the content of `r1`. The dependency is a false one, since the content of `r3` always is zero, regardless of the contents of `r1`.

6.2.4.2 Control Dependencies

These candidate relaxations have the following syntax:

$$\mathit{Ctrl}(s \mid d)(R \mid W)$$

A control dependency is implemented with:

- a useless compare and branch sequence when the target is a write;
- a useless compare and branch sequence followed (for PowerPC) by an `isync` instruction when the target is a read.

Consider for example the following PowerPC assembly code:

```
(1) lwz r1,0(r5)
(2) cmpwi r1,0
(3) bne L0
(4) stw r2,0(r6)
L0:
```

Suppose that the register `r5` initially holds the address x, and the register `r6` the address y. The instructions `lwz r1,0(r5)` at line (1) and `stw r2,0(r6)` at line (4) correspond to a CtrldW candidate relaxation. Indeed, the first one is a read and the second one a write. They are to distinct locations, and in control dependency, since they are separated by a compare and branch sequence, composed of `cmpwi r1,0` at line (2) and `bne L0` at line (3).

In both cases, we assume that dependencies are not erased by the assembler or the hardware, as the result of an optimisation.

6.2.5 Composite Candidate Relaxations

We call *composite candidate relaxations* any sequence of candidate relaxations. Their syntax is as follows (with r_1 and r_2 being candidate relaxations):

$$[r_1, r_2, \ldots]$$

We use composite candidate relaxations to specify *e.g. cumulativity candiate relaxations*. For example, we write ACSyncsRR for an A-cumulativity candidate relaxation for two reads to the same location separated in program order by a `sync` barrier. We specify this candidate relaxation by the composite candidate relaxation [Rfe, SyncsRR]. Similarly, we write BCLwSyncdWR for a B-cumulativity candidate relaxation for a read event followed by a write to a different location, separated in program order by a `lwsync`. We specify this candidate relaxation by the composite candidate relaxation [LwSyncdWR, Rfe].

6.3 A Preliminary Power Model

Our rather liberal interpretation of the documentation led us to a first informal Power model. We considered all the examples exposed in the documentation as guarantees: *e.g.* we considered that the Power architecture preserves globally the read-read pairs in program order, as long as there is a data dependency between them (see Sec. 6.1.3.1).

We summarise this informal model in Fig. 6.3, where:

- an underscore _ depicts all the possibilities: *e.g.* Ws_ stands for both Wsi and Wse, and Dp_R for both DpdR and DpsR;
- "a pair" means a pair of events in program order;
- "a pair separated by [something]" means the events are in program order, and there is [something] between them in program order;
- unless otherwise stated, the relation is valid regardless of the directions and locations of the events;
- $m_1 \overset{\text{sync}}{\to} m_2$ (resp. $m_1 \overset{\text{lwsync}}{\to} m_2$) means the instructions i_1 and i_2 associated to the events m_1 and m_2 are separated in program order by a `sync` (resp. `lwsync`) barrier (we formalise this notion in Sec. 8.2.1.3, see infra).

From a communication perspective, Power allows store buffering and relaxes the atomicity of stores. From a preserved program order point of view, Power preserves only the read-read and read-write pairs in dependency, whether data or control. From a barrier point of view, Power features cumulative barriers. Such barriers are a way to compensate for the lack of store atomicity, as we will see in Sec. 10.3.3.

We explain how we tested this model in the next two chapters.

Candidate Relaxation	Comment	Ref. in [pow09]
Ws_	write serialisation	p. 657, 1st col, last §
Fr_	from-read map	p. 661, 2nd col, penultimate §
PosRW (= Fri)	any load-store pair to the same location	p. 661, 2nd col, penultimate §
PosWW (= Wsi)	any write-write pair to the same location	p. 657, 1st col, last §
Dp_R	any load-load pair in data dependency	p. 660, 1st col, last bullet
Dp_W	any load-store pair in data dependency	p. 661, 1st col, 1st §
Ctrl_R	any load-load pair separated by a compare and branch sequence followed by an `isync`	p. 661, 1st col, 2nd §
Ctrl_W	any load-store pair separated by a compare and branch sequence	p. 661, 1st col, 1st §
Sync___	any pair separated by a `sync`	p. 700, 2nd col, 1st bullet
LwSync_d_1d_2 w. $(d_1, d_2) \neq (W, R)$	any load-load, load-store or store-store pair separated by a `lwsync`	p. 700, 1st col, L=1 case
ACSync___	$m_1 \overset{\text{rfe}}{\rightarrow} m_2 \overset{\text{sync}}{\rightarrow} m_3$	p. 700, 1st col, last line
BCSync___	$m_1 \overset{\text{sync}}{\rightarrow} m_2 \overset{\text{rfe}}{\rightarrow} m_3$	p. 700, 1st col, last line
ACLwSync_d_1d_2 w. $(d_1, d_2) \neq (R, R)$	$m_1 \overset{\text{rfe}}{\rightarrow} m_2 \overset{\text{lwsync}}{\rightarrow} m_3$, with $m_3 \notin \mathbb{R}$	p. 700, 1st col, last line
BCLwSync_d_1d_2 w. $(d_1, d_2) \neq (W, W)$	$m_1 \overset{\text{lwsync}}{\rightarrow} m_2 \overset{\text{rfe}}{\rightarrow} m_3$, with $m_1 \notin \mathbb{W}$	p. 700, 1st col, last line

Figure 6.3: Safe Candidate Relaxations According to the Documentation

Chapter 7

Diy, A Testing Tool

When she received her brand new computer, Monique was confused. Nowhere in the forest of accompanying leaflets could she find the operating instructions she demanded. How would she be able to find the minute details of her computer's MEMORY MODEL? Were reads reorderings allowed? Was the read-from relation global? And what tests could she devise to tackle the low-level questions every perfect housewife should ask herself?

Little did she know that the answer laid within reach of her hard-working yet impeccably manicured hands, on the d.i.y. web site!

We present our diy (*do it yourself*) tool, which computes litmus tests in x86 or Power assembly code by generating violations of SC, *i.e.* cycles in $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$. A diy tutorial is available at `http://diy.inria.fr`.

7.1 Litmus Tests

We use litmus tests throughout this document to highlight which behaviours are allowed and which are forbidden on a given architecture.

7.1.1 Highlighting Relaxations

A given test may exhibit its specified outcome for several different reasons. Consider for example the test of Fig. 7.1. We already discussed in Sec. 4.2.4.2 why the outcome of this test may be exhibited. We recall the reasoning here. The specified outcome may be revealed by an execution such as the one we depict in Fig. 7.2. Suppose that each location and register initially hold 0. If `r1` holds 1 on P_0 in the end, the read (a) has read its value from the write (e) on P_2, hence $(e) \xrightarrow{\text{rf}} (a)$. On the contrary, if `r2` holds 0 in the end, the read (b) has read its value from the initial state, thus before the write (f) on P_3, hence $(b) \xrightarrow{\text{fr}} (f)$. Similarly, we have $(f) \xrightarrow{\text{rf}} (c)$ from `r2` holding 1 on P_1, and $(d) \xrightarrow{\text{fr}} (e)$ from `r1` holding 0 on P_1. Hence, if the specified outcome is observed, it ensures that at least one execution of the test contains the cycle depicted in Fig. 7.2, written here in terms of the candidate relaxations presented in Chap. 6:

$$(a) \xrightarrow{\text{PodRR}} (b) \xrightarrow{\text{Fre}} (f) \xrightarrow{\text{Rfe}} (c) \xrightarrow{\text{PodRR}} (d) \xrightarrow{\text{Fre}} (e) \xrightarrow{\text{Rfe}} (a)$$

The outcome may be exhibited for two reasons: either the external $\xrightarrow{\text{rf}}$ are not global on the machine running the test, or the read-read pairs ($(a) \xrightarrow{\text{po}} (b)$ on P_0 and $(c) \xrightarrow{\text{po}} (d)$ on P_1) are not preserved in the program order. Therefore, when running this test, we cannot decide which weakness of the memory model is responsible for the outcome to be exhibited.

7.1.2 Exercising One Relaxation at a Time

So as to make the analysis of the testing results feasible, we focus on tests which exercise a unique weakness of the memory model at a time. Hence, if the outcome of a given test is exhibited, we know that the feature we tested is used by the machine on which we ran the test.

For example, suppose that we modify the test of Fig. 7.1 and impose dependencies between the pairs of reads on P_0 and P_1, so that these dependencies are global, *e.g.* by being included in $\xrightarrow{\text{ppo}}$. We give in Fig. 7.5 the **iriw** test, written in PowerPC assembly, and modified to impose such global

iriw

P_0	P_1	P_2	P_3
$(a)\,\texttt{r1} \leftarrow \texttt{x}$	$(c)\,\texttt{r2} \leftarrow \texttt{y}$	$(e)\,\texttt{x} \leftarrow \texttt{1}$	$(f)\,\texttt{y} \leftarrow \texttt{2}$
$(b)\,\texttt{r2} \leftarrow \texttt{y}$	$(d)\,\texttt{r1} \leftarrow \texttt{x}$		

Observed? `0:r1=1`; `0:r2=0`; `1:r2=2`; `1:r1=0`;

Figure 7.1: Study of **iriw**

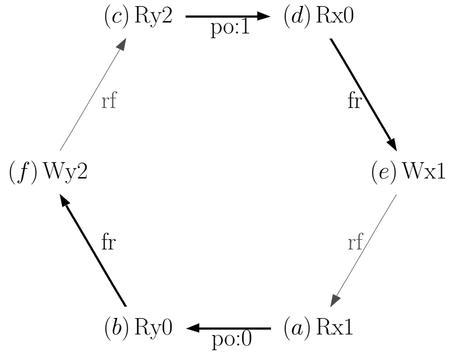


Figure 7.2: A non-SC execution of **iriw**

dependencies to the pairs of reads. In order to do so, we use a `xor` between the loads (a) and (b) on P_0 and (c) and (d) on P_1. Hence (a) is in data dependency with (b), and so is (c) with (d).

In this case the only reason why the specified outcome may arise is the non-globality of external read-from maps. Hence, the test of Fig. 7.5 is significant to check whether an architecture relaxes the atomicity of stores.

7.2 Cycles as Specifications of Litmus Tests

We want to be able to generate such significant tests, automatically and systematically. We noticed that the interesting behaviours of a litmus test can be characterised by cycles formed of the relations of our model.

7.2.1 Automatic Test Generation

For example, as we exposed above, the outcome of Fig. 7.1 leads to the $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$ cycle depicted in Fig. 7.2. Conversely, the **iriw** test of Fig. 7.1 can be built from the sequence $\xrightarrow{\text{rfe}};\xrightarrow{\text{po}};\xrightarrow{\text{fre}};\xrightarrow{\text{rfe}};\xrightarrow{\text{po}};\xrightarrow{\text{fre}}$ interpreted as a cycle. The computed outcome ensures—as we exposed in Sec. 7.1.1—that the input cycle appears in at least one of the execution witnesses of the test. If the outcome is observed, then at least one subsequence in the cycle is not global, *i.e.* not in $\xrightarrow{\text{ghb}}$: in the case of Fig. 7.1, either the $\xrightarrow{\text{po}}$ or the $\xrightarrow{\text{rfe}}$ relations may not be included in $\xrightarrow{\text{ghb}}$.

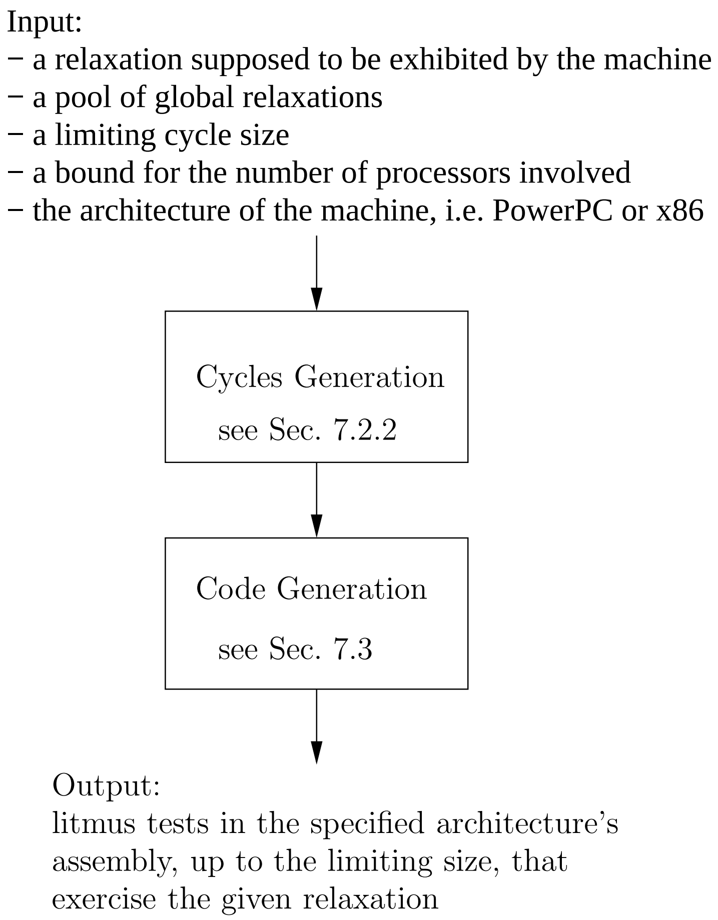

Input:
− a relaxation supposed to be exhibited by the machine
− a pool of global relaxations
− a limiting cycle size
− a bound for the number of processors involved
− the architecture of the machine, i.e. PowerPC or x86
Cycles Generation
see Sec. 7.2.2
Code Generation
see Sec. 7.3
Output:
litmus tests in the specified architecture's
assembly, up to the limiting size, that
exercise the given relaxation

Figure 7.3: Overview of Diy

To generate automatically litmus tests that exercise relations specified by the user, we wrote the diy testing tool. We give in Fig. 7.3 an overview of the tool. When given a certain sequence of relations, diy produces tests such that one of their executions contains at least one occurrence of the given sequence. Hence, if we want to check whether the external read-from maps are relaxed on a given machine, we specify $\xrightarrow{\text{rfe}}$ to be relaxed to diy, following the concrete syntax we give in Sec. 6.2. diy then produces cycles containing at least one occurrence of the specified relation (*e.g.* $\xrightarrow{\text{rfe}}$ in our case), up to a certain size, as exposed in Sec. 7.2.2. Each cycle is then parsed to produce a litmus test exercising the given relaxation on the specified architecture, following the code generation algorithm presented in Sec. 7.3.

```
#rfe PPC conf file
-arch PPC
-nprocs 4
-size 6
-name rfe
-safe Fre DpdR
-relax Rfe
```

Figure 7.4: Example Configuration File for Diy

7.2.2 Cycles Generation

diy needs to be specified which candidate relaxations are considered global and which are not. When specified a pool of safe candidate relaxations, a single non-global relaxation (expected to be relaxed), and a size n (*i.e.* the number of candidate relaxations arrows in the cycle, *e.g.* 6 for the **iriw** test of Fig. 7.1), diy generates cycles up to size n that contains at least one occurrence of the non-global candidate relaxation. If no non-global candidate relaxation is specified, diy generates cycles up to size n that contain the specified global candidate relaxations.

We do not generate all these cycles: we indeed eliminate some sequences of candidate relaxations. In particular, we eliminate a sequence of two candidate relaxations when the target of the first one is incompatible with the source of the second one: for example, we eliminate a sequence $\xrightarrow{\text{Rfe}};\xrightarrow{\text{Rfe}}$ because the target of the first $\xrightarrow{\text{Rfe}}$ is a read, whereas the source of the second one is a write. We also eliminate any sequence of candidate relaxations included in the program order (whether two events in program order or separated by a fence). This reduces the number of cycles to generate, and in any case, such sequences of candidate relaxations can be specified by using composite candidate relaxations (see Sec. 6.2.5).

We give in Fig. 7.4 an example of *configuration file* that diy takes as an argument to generate tests. This configuration file forces diy to generate tests in PowerPC assembly up to 4 processors, as specified by the `-arch PPC` and `-nprocs 4` arguments, so that the number of relations involved in the generated cycles is 6 at most, because of the `-size 6` argument. Moreover, the candidate relaxations Fre (external from-read map) and DpdR (data dependency between two reads from distinct locations) are considered global, and Rfe is considered relaxed, as specified by the `-safe Fre DpdR` and `-relax Rfe` arguments. Finally, all the tests generated by diy running on this configuration file will have the prefix rfe in their name, followed by a fresh number, as specified by the `-name rfe` argument.

When the generation of cycles is over, diy computes litmus tests from those cycles, as detailed in the following.

7.3 Code Generation

diy interprets a sequence of candidate relaxations as a cycle from which it computes a litmus test or fails. The final condition is a conjunction of equalities on the values hold by registers and memory locations in the final state. This condition ensures that at least one of the execution witnesses of this test includes a cycle compliant with the input sequence, following the principle exposed in Sec. 7.1.2.

7.3.1 Algorithm

Test generation performs the following successive steps:

1. We map the edges sequence to a circular double-linked list. The cells represent memory events, with direction, location, and value fields. An additional field records the edge starting from the event. This list represents the *input cycle* and appears in at least one of the execution witnesses of the produced test.

2. A linear scan sets the directions of the events, by comparing each target direction with the following source direction. When they are equal, the in-between cell direction is set to the common value; otherwise (*e.g.* Rfe; Rfe), the generation fails.

3. We pick an event e which is the target of a candidate relaxation specifying a location change. If there is none, the generation fails. Otherwise, a linear scan starting from e sets the locations. At the end of the scan, if e and its predecessor have the same location (*e.g.* $\xrightarrow{\text{Rfe}} e \xrightarrow{\text{PodRW}}$), the generation fails, since we picked e to correspond to a location change.

4. We cut the input cycle into maximal sequences of events with the same location, each being scanned *w.r.t.* the cycle order: we give the value 1 to the first write in this sequence, 2 to the second one, *etc.* Thus the writes' values reflect the write serialisation for the specified location.

5. We define *significant reads* as the sources of $\xrightarrow{\text{fr}}$ edges and the targets of $\xrightarrow{\text{rf}}$ edges. We associate each significant read with the write on the other side of the edge. In the $\xrightarrow{\text{rf}}$ case, the value of the read is the one of its associated write. In the $\xrightarrow{\text{fr}}$ case, the value of the read is the value of the predecessor of its associated write in $\xrightarrow{\text{ws}}$, *i.e.* by construction the value of its associated write minus 1 (see step 4). Non-significant reads do not appear in the test condition.

6. We cut the cycle into maximal sequences of events from the same processor, each being scanned, generating load instructions to (resp. stores from) fresh registers for reads (resp. writes). We insert some

code implementing a dependency in front of events targeting $\xrightarrow{\text{dp}}$ and the appropriate barrier instruction for events targeting $\xrightarrow{\text{fenced}}$ edges. We build the initial state at this step: stores and loads take their addresses from fresh registers, and their content (an address to a memory location) is defined in the initial state. Part of the final condition is also built: for any significant read with value v resulting in a load instruction to register r, we add the equality $r = v$.

7. We complete the final condition to characterise write serialisations. The write serialisation for a given location x is defined by the sequence of values 0 (initial value of x), …, n, where n is the last value allocated for location x at step 4. If n is 0 or 1 then no addition to the final condition needs to be performed, because the write serialisation is either a singleton or a pair. If n is 2, we add the equality $x = 2$. Otherwise ($n > 2$), we add an *observer* to the program, *i.e.* we add a thread performing n loads from x to registers `r1`, …, `rn` and add the equalities $\texttt{r1} = 1 \wedge \ldots \wedge \texttt{rn} = n$ to the final condition.

7.3.2 Example

We show here how to generate a Power litmus test from a given cycle of candidate relaxations by an example. We write $\_$ for the information not yet set by diy: $\_\_\_$ is an undetermined event, $W\_\_$ a write with yet unset location and value, and $Rx\_$ a read from x with undetermined value.

1. Consider *e.g.* the input cycle, issued by diy's cycles generation phase, with the input being the configuration file given in Fig. 7.4:

$$(a)\_\_\_ \xrightarrow{\text{Rfe}} (b)\_\_\_ \xrightarrow{\text{DpdR}} (c)\_\_\_ \xrightarrow{\text{Fre}} (d)\_\_\_ \xrightarrow{\text{Rfe}} (e)\_\_\_ \xrightarrow{\text{DpdR}} (f)\_\_\_ \xrightarrow{\text{Fre}} (a)$$

2. A linear scan sets the directions from the edges. Observe *e.g.* the last edge; $\xrightarrow{\text{Fre}}$ requires a R source and a W target:

$$(a)W\_\_ \xrightarrow{\text{Rfe}} (b)R\_\_ \xrightarrow{\text{DpdR}} (c)R\_\_ \xrightarrow{\text{Fre}} (d)W\_\_ \xrightarrow{\text{Rfe}} (e)R\_\_ \xrightarrow{\text{DpdR}} (f)R\_\_ \xrightarrow{\text{Fre}} (a)$$

3. As $\xrightarrow{\text{DpdR}}$ specifies a location change, we pick (c) to be the first event and rewrite the cycle as:

$$(c)R\_\_ \xrightarrow{\text{Fre}} (d)W\_\_ \xrightarrow{\text{Rfe}} (e)R\_\_ \xrightarrow{\text{DpdR}} (f)R\_\_ \xrightarrow{\text{Fre}} (a)W\_\_ \xrightarrow{\text{Rfe}} (b)R\_\_ \xrightarrow{\text{DpdR}} (c)$$

We set the locations starting from (c), with a change of location *e.g.* between (e) and (f) since $\xrightarrow{\text{DpdR}}$ specifies a location change:

$$(c)Rx\_ \xrightarrow{\text{Fre}} (d)Wx\_ \xrightarrow{\text{Rfe}} (e)Rx\_ \xrightarrow{\text{DpdR}} (f)Ry\_ \xrightarrow{\text{Fre}} (a)Wy\_ \xrightarrow{\text{Rfe}} (b)Ry\_ \xrightarrow{\text{DpdR}} (c)$$

```
{ 0:r2=y; 0:r5=x; 1:r2=x; 2:r2=x; 2:r5=y; 3:r2=y; }

 P0               | P1                | P2                 | P3
(b) lwz r1,0(r2)  |     li r1,1       | (e) lwz r1,0(r2)   |    li r1,1
    xor r3,r1,r1  | (d) stw r1,0(r2)  |     xor r3,r1,r1   |(a) stw r1,0(r2)
(c) lwzx r4,r3,r5 |                   | (f) lwzx r4,r3,r5  |

exists (0:r1=1 /\ 0:r4=0 /\ 2:r1=1 /\ 2:r4=0)
```

Figure 7.5: **iriw** with dependencies in Power assembly

4. We cut the input cycle into maximal sequences of events with the same location (*i.e.* $(c)(d)(e)$ and $(f)(a)(b)$), each being scanned *w.r.t.* the cycle order. The values then reflect the write serialisation order for the specified location:

$$(c)Rx\_\xrightarrow{\text{Fre}}(d)Wx1\xrightarrow{\text{Rfe}}(e)Rx\_\xrightarrow{\text{DpdR}}(f)Ry\_\xrightarrow{\text{Fre}}(a)Wy1\xrightarrow{\text{Rfe}}(b)Ry\_\xrightarrow{\text{DpdR}}(c)$$

5. All the reads are significant here; we set their values according to step 5:

$$(c)Rx0\xrightarrow{\text{Fre}}(d)Wx1\xrightarrow{\text{Rfe}}(e)Rx1\xrightarrow{\text{DpdR}}(f)Ry0\xrightarrow{\text{Fre}}(a)Wy1\xrightarrow{\text{Rfe}}(b)Ry1\xrightarrow{\text{DpdR}}(c)$$

6. We generate the litmus test given in Fig. 7.5 for Power according to 6 and 7. For example on P_0, we add a `xor` instruction between the instructions `lwz r1,0(r2)` and `lwzx r4,r3,r5` associated with the events (b) and (c) to implement the dependency required by the $\xrightarrow{\text{DpdR}}$ relation between them. The events (d) and (e), associated respectively to `stw r1,0(r2)` on P_1 and `lwz r1,0(r2)` on P_2, are specified in the cycle to be in $\xrightarrow{\text{rfe}}$. Hence, we specify in the final state that the register `r1` on P_2 holds finally 1. Indeed the store associated with (d) writes 1 into the address x addressed by `r2` on P_1 , since the contents of the register `r1` on P_1 is 1 (because of the preceding `li r1, 1` instruction). Since $(d) \xrightarrow{\text{rfe}} (e)$, the load associated with (e) on P_2 reads the value 1 from the address x addressed by `r2`, and writes 1 into the register `r2`.

The test in Fig. 7.5 is a Power implementation of **iriw** [BA] with dependencies. It can be obtained by running diy on the configuration file given in Fig. 7.4. diy recovers indeed classical tests, such as **rwc** [BA], given in Fig. 7.6(a). As one can deduce from the execution given in Fig. 7.6(b), this test can be obtained from the cycle $\xrightarrow{\text{Rfe}};\xrightarrow{\text{PodRR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{PodWR}};\xrightarrow{\text{Fre}}$.

7.4 A First Testing Example: x86-TSO

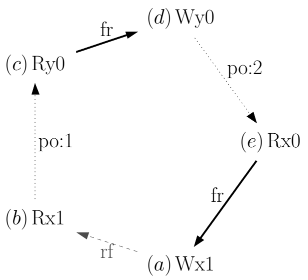
x86 has a TSO model [OSS]. As we saw in Sec. 4.2.2.2, this means that the internal read-from maps are not global, and that the write-read pairs in program order may not be preserved. All the other relations are global.

rwc

P_0	P_1	P_2
(a) `x` ← `1`	(b) `r1` ← `x`	(d) `y` ← `1`
	(c) `r2` ← `y`	(e) `r3` ← `x`

Observed? `1:r1=1; 1:r2=0; 2:r3=0`

(a) The **rwc** test



(b) An Execution Witness

Figure 7.6: The **rwc** Test and a Candidate Execution

7.4.1 A Guided Diy Run

We give here a step-by-step protocol to generate and run tests to check whether the reordering of a write followed by a read in program order is allowed on a x86 machine, *i.e.* if the PodWR candidate relaxation is actually relaxed on a x86-TSO machine.

7.4.1.1 Generating a Test for Testing the PodWR Relaxation

We want to generate at least one test exercising the PodWR candidate relaxation, *e.g.* the classical litmus test given in Fig. 3.1(a). In the execution depicted in Fig. 3.1(b), there is a cycle $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (a)$. The event (a) (resp. (c)) is a write, since it is the target of an $\xrightarrow{\text{fr}}$ arrow. Similarly, (b) (resp. (d)) is a read since it is the source of an $\xrightarrow{\text{fr}}$ arrow. Moreover, the test of Fig. 3.1(a) involves two processors, therefore the $\xrightarrow{\text{fr}}$ arrows are in fact external from-read maps. We suppose that $\xrightarrow{\text{fr}}$ is always global, therefore the only possible relaxation is PodWR.

Hence this test can be generated from the cycle $\xrightarrow{\text{PodWR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{PodWR}};\xrightarrow{\text{Fre}}$, specifying the architecture to be x86, and the number of processors to be 2. The command:

```
$ diy -arch X86 -nprocs 2 -safe Fre -relax PodWR -name classic
```

produces a test, `classic000.litmus`, testing the PodWR candidate relaxation:

```
Generator produced 1 tests
Relaxations tested: {PodWR}
```

The test `classic000.litmus` is the test of Fig. 3.1 implemented in x86 assembly:

```
% cat classic000.litmus
X86 classic000
"Fre PodWR Fre PodWR"
{ x=0; y=0; }
 P0          | P1          ;
 MOV [y],$1  | MOV [x],$1  ; #(a)Wy1  | (c)Wx1
 MOV EAX,[x] | MOV EAX,[y] ; #(b)Rx0  | (d)Ry0
exists (0:EAX=0 /\ 1:EAX=0)
```

7.4.1.2 Running This Test Against Hardware

Now we want to run this test against our machine, which should of course be an x86 one, with at least two cores.

The command:

```
$ litmus classic000.litmus
```

runs our test against the hardware and produces an output similar to the following one:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Results for classic000.litmus %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
X86 classic000
"Fre PodWR Fre PodWR"

{ x=0; y=0; }

 P0          | P1          ;
 MOV [y],$1  | MOV [x],$1  ;
 MOV EAX,[x] | MOV EAX,[y] ;

exists (0:EAX=0 /\ 1:EAX=0)
Generated assembler
        _litmus_P0_0_: movl $1,(%rcx)
        _litmus_P0_1_: movl (%rsi),%eax
        _litmus_P1_0_: movl $1,(%rsi)
        _litmus_P1_0_: movl $1,(%rsi)
        _litmus_P1_1_: movl (%rcx),%eax

Test classic000 Allowed
Histogram (4 states)
34    :>0:EAX=0; 1:EAX=0;
499911:>0:EAX=1; 1:EAX=0;
499805:>0:EAX=0; 1:EAX=1;
```

```
#rfi x86 conf file
-arch X86
-nprocs 4
-size 6
-name rfi
-safe PosR* PodR* PodWW PosWW
  Rfe Wse Fre
  FencedsWR FenceddWR
-relax Rfi
```

Figure 7.7: x86 Configuration File For Rfi Relaxation

```
250   :>0:EAX=1; 1:EAX=1;
Ok

Witnesses
Positive: 34, Negative: 999966
Condition exists (0:EAX=0 /\ 1:EAX=0) is validated
Hash=eb447b2ffe44de821f49c40caa8e9757
Time classic000 0.60
...
```

This output first reproduces the test litmus ran, followed by the actual x86 assembly that was run one million times. The code is followed by a histogram, which gives the 4 states of the memory that were observed while running this test. The first one is the one of interest to us, as it was the non-SC outcome specified by diy. It was exhibited 34 times on the hardware: therefore litmus wrote `Ok`, which means we observed the PodWR candidate relaxation to be exhibited.

The same test, and additional ones, can be obtained by running directly:

```
$ diy -conf podwr.conf
```

where `podwr.conf` is the configuration file for the PodWR candidate relaxation given in Fig. 7.8.

7.4.2 Configuration Files

Finally, we give the configuration files to test a x86 machine. In Fig. 7.7, we give a configuration file to generate tests exercising the internal read-from maps, as specified by the `-relax Rfi` argument. In Fig. 7.8, we give a configuration file to generate tests exercising the write-read pairs in program order and to distinct locations, as specified by the `-relax PodWR` argument. There is no need to generate tests for the PosWR candidate relaxation, since such pairs are handled by the Rfi tests. In Fig. 7.9, we give a configuration

```
#podwr x86 conf file
-arch X86
-nprocs 4
-size 6
-name podwr
-safe Fre
-relax PodWR
```

Figure 7.8: x86 Configuration File For PodWR Relaxation

```
#safe x86 conf file
-arch X86
-nprocs 4
-size 6
-name safe
-safe PosR* PodR* PodWW PosWW
  Rfe Wse Fre
  FencedsWR FenceddWR
```

Figure 7.9: x86 Configuration File For Safe Relaxations

file to generate tests exercising all the candidate relaxations that we consider to be safe in *TSO*.

Chapter 8

A Power Model

8.1 The Phat Experiment

With the informal model of Sec. 6.3 in mind, we ran The Phat Experiment from December 2009 to January 2010, as a case study for the diy tool presented in Chap. 7. We tested 3 Power machines with our diy tool. We present here the experimental results. More details can be found online at `http://diy.inria.fr/phat`.

8.1.1 Relaxations Observed on squale, vargas and hpcx

We used diy to generate 800 Power tests and run them up to 10^{12} times each on 3 machines: `squale`, a 4-processor Power G5 running Mac OS X; `hpcx`, a Power 5 with 16 processors per node and `vargas`, a Power 6 with 32 processors per node, both of them running AIX.

We ran the tests supposed to, according to the informal model presented in Sec. 6.3, exhibit relaxations. We observed all of them at least on one machine, except PodRW, which does not contradict our model. We give in Fig. 8.1 the number of times the outcome was observed (where M stands for million). For each relaxation observed on a given machine, we write the highest number of outcomes. When a candidate relaxation is not observed, we write the total of outcomes: thus we write *e.g.* 0/16725M for PodRR on `vargas`.

For a given candidate relaxation, we generated tests with diy by writing a simple configuration file setting its relax list to this candidate relaxation, and some of the candidate relaxations that we considered to be safe (see Fig. 6.3).

The PodRW relaxation did not exhibit itself in convincing ways. But the documentation does not specify this candidate relaxation to be safe, therefore we still consider it to be relaxed.

8.1.2 Safe Relaxations

Following our informal model, we assumed that the candidate relaxations of Fig. 6.3 were global and tested this assumption by computing *safe* tests in which the input cycles only include candidate relaxations that we supposed global, *e.g.* $\xrightarrow{\text{SyncdWW}};\xrightarrow{\text{Wse}};\xrightarrow{\text{SyncdWR}};\xrightarrow{\text{Fre}}$.

For each machine, we observed the number of runs required to exhibit the least frequent relaxation (*e.g.* 32 million for Rfe on `vargas`), and ran the safe tests at least 20 times this number. The outcomes of the safe tests have not been observed on `vargas` and `squale`, which increases our confidence in the safe set we assumed.

Yet, `hpcx` exhibits non-*SC* behaviours for some A-cumulativity tests, including classical ones [BA] like **iriw** with `sync` instructions on P_0 and

Relaxation	Definition	`hpcx`	`squale`	`vargas`
PosRR	$r_\ell \xrightarrow{\text{po}} r'_\ell$	2/40M	3/2M	0/4745M
PodRR	$r_\ell \xrightarrow{\text{po}} r'_{\ell'}$	2275/320M	12/2M	0/16725M
PodRW	$r_\ell \xrightarrow{\text{po}} w'_{\ell'}$	0/6000M	0/6000M	0/6000M
PodWW	$w_\ell \xrightarrow{\text{po}} w'_{\ell'}$	2029/4M	2299/2M	2092501/32M
PodWR	$w_\ell \xrightarrow{\text{po}} r'_{\ell'}$	51085/40M	178286/2M	672001/32M
Rfi	$\xrightarrow{\text{rfi}}$	7286/4M	1133/2M	145/32M
Rfe	$\xrightarrow{\text{rfe}}$	177/400M	0/1776M	9/32M
LwSyncsWR	$w_\ell \xrightarrow{\text{lwsync}} r'_\ell$	243423/600M	2/40M	385/32M
LwSyncdWR	$w_\ell \xrightarrow{\text{lwsync}} r'_{\ell'}$	103814/640M	11/2M	117670/32M
ACLwSyncsRR	$w_\ell \xrightarrow{\text{rfe}} r'_\ell \xrightarrow{\text{lwsync}} r''_\ell$	11/320M	0/960M	1/21M
ACLwSyncdRR	$w_\ell \xrightarrow{\text{rfe}} r'_\ell \xrightarrow{\text{lwsync}} r''_{\ell'}$	124/400M	0/7665M	2/21M
BCLwSyncsWW	$w_\ell \xrightarrow{\text{lwsync}} w'_\ell \xrightarrow{\text{rfe}} r''_\ell$	68/400M	0/560M	2/160M
BCLwSyncdWW	$w_\ell \xrightarrow{\text{lwsync}} w'_{\ell'} \xrightarrow{\text{rfe}} r''_{\ell'}$	158/400M	0/11715M	1/21M

Figure 8.1: Selected Results of the Diy Experiment Matching Our Model

Cycle	`hpcx`	In [BA]
Rfe SyncdRR Fre Rfe SyncdRR Fre	2/320M	iriw
Rfe SyncdRR Fre SyncdWR Fre	3/400M	rwc
DpdR Fre Rfe SyncsRR DpdR Fre Rfe SyncsRR	1/320M	
Wse LwSyncdWW Wse Rfe SyncdRW	1/800M	
Wse SyncdWR Fre Rfe LwSyncdRW	1/400M	

Figure 8.2: Anomalies Observed on Power 5

P_1. These results are in contradiction with our model. We summarise these contradictions in Fig. 8.2.

We understand that this is due to an erratum in the Power 5 implementation. IBM is providing a software workaround, replacing the `sync` barrier by a short code sequence [Personal Communication], and our testing suggests that this does regain *SC* behaviour for the examples in question (e.g. with 0/4e10 non-*SC* results for **iriw**). We understand also that Power 6 is not subject to the erratum, which is consistent with our testing on `vargas`.

8.2 Overview of Our Model

We now instantiate the formalism of Sec. 3.1 for Power, in the light of the experiment described in Sec. 8.1.

8.2.1 Additional Formalism

To do so, we have to extend our formalism a bit, as follows.

8.2.1.1 Register Events

We first add *register events* to reflect register accesses [SSZN$^+$]. Loads and stores now yield additional register events, as depicted in Fig. 8.3.

For example, consider two registers `r1` and `r2`, such that `r1` initially holds the value 0, `r2` initially holds an address x, and x holds the value 1. In this case, an instruction `lwz r1,0,r2` creates a read event $R\texttt{r2}x$ from register `r2`, with label (b) in Fig. 8.3. This event reads the address x in `r2`; this leads to a read event $Rx1$ from x labelled (a) in Fig. 8.3. The event (a) was previously the only event we considered. Finally, the value read from x by the event (a) being 1, the `lwz r1,0,r2` creates a write event $W\texttt{r1}1$ to register `r1` with value 1, labelled (c) in Fig. 8.3.

Similarly, consider two registers `r1` and `r2`, such that `r1` initially holds the value 1, `r2` initially holds an address x, and x holds the value 0. In this case, an instruction `stw r1,0,r2` creates a read event $R\texttt{r2}x$ from register `r2`, with label (c) in Fig. 8.3. This event reads the address x in `r2`. In

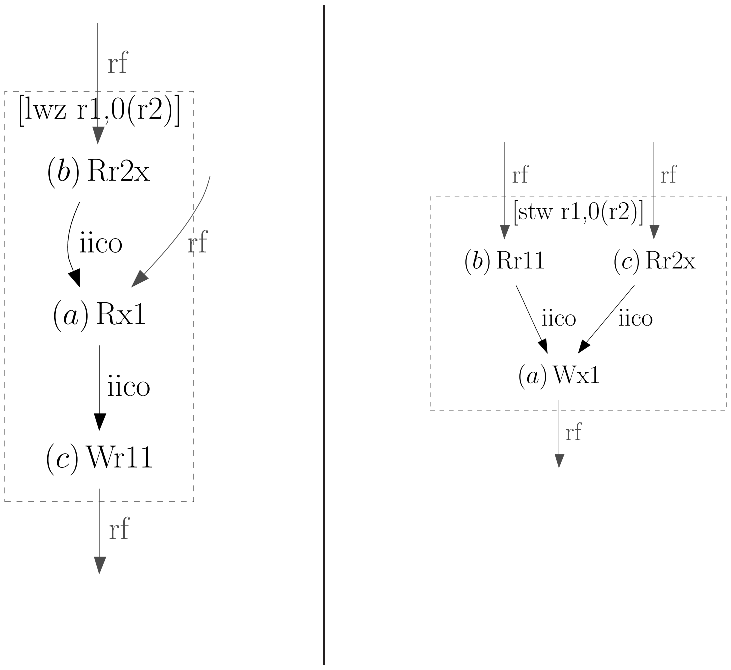


Figure 8.3: Semantics of `lwz` and `stw`

parallel, the load creates a read event R`r1`1 from `r1`, reading 1, labelled (c) in Fig. 8.3. Finally, the value read from `r1` by the event (b) being 1, the `stw r1,0,r2` creates a write event $Wx1$ to x with value 1, labelled (a) in Fig. 8.3. The event (a) previously was the only event we considered.

Intra-Instruction Causality An execution witness now includes an additional *intra-instruction causality* relation $\xrightarrow{\text{iico}}$.

For example, executing the load `lwz r1, 0(r2)`—which semantics is given in Fig. 8.3 (`r2` holding the address of a memory location x containing 1)—creates three events (a) R`r2`x, (b) Rx1 and (c) W`r1`1, such that $(a) \xrightarrow{\text{iico}} (b) \xrightarrow{\text{iico}} (c)$. The $\xrightarrow{\text{iico}}$ relation between (a) and (b) indicates that the load instruction has to perform the read (a) from `r2` before the read (b) from x. Before reading x from `r2` via the event (a), the address x is undetermined. Similarly, (b) and (c) are related by $\xrightarrow{\text{iico}}$, since the write (c) determines the value it has to write into `r1` from the value read by (b).

Stores are less constrained, as depicted in Fig. 8.3. Indeed the read events (b) and (c) may be performed independently. However, the write event (a) determines its target x and its value 1 from the reads (c) and (b) respectively, hence $(b) \xrightarrow{\text{iico}} (a)$ and $(c) \xrightarrow{\text{iico}} (a)$.

```
PPC ctrl
{
0:r5=x; 0:r6=y;0:r2=1;
x=0; y=0;
}
P0                 ;
(1) lwz r1,0(r5) ;
(2) cmpwi r1,0   ;
(3) bne L0       ;
(4) stw r2,0(r6) ;
L0:                ;
exists(y=1)
```

Figure 8.4: A Test For Control Dependency in PowerPC Assembly

Read-From Map Naturally, $\xrightarrow{\text{rf}}$ now also relates register events: we write $\xrightarrow{\text{rf-reg}}$ the subrelation of $\xrightarrow{\text{rf}}$ relating register stores to register loads that read their values.

8.2.1.2 Commit Events

We add *commit events* in order to express branching decisions. We write $\mathbb{C}$ for the set of commits, and c for an element of $\mathbb{C}$.

Consider for example the test given in Fig. 8.4, written in PowerPC assembly, which corresponds to the piece of code given in Sec. 6.2.4.2. Suppose that the register `r5` initially holds the address x, and the register `r6` the address y.

The `lwz r1,0(r5)` at line (1) and the `stw r2,0(r6)` at line (4) are separated at line (2) by a compare instruction, written `cmpwi r1,0`, which influences the taking of the following branch, written `bne L0` (line (3)).

In the execution of this test, given in Fig. 8.5, the `lwz r1,0(r5)` leads to the read event $Rx0$ from x labelled (a). The `stw r2,0(r6)` leads to the write event $Wy1$ to y labelled (b). These two instructions are separated by a branch. This is depicted in the execution we give here by the commit event labelled (h). Hence the read (a) from x and the write (b) to y are in control dependency, as depicted by the $\xrightarrow{\text{ctrl}}$ arrow between them.

8.2.1.3 Barriers Events

We add *barrier events* in order to indicate the presence of a barrier in the code. We handle three barrier instructions : `isync`, `sync` and `lwsync`. Thus, we distinguish the corresponding events by the eponymous predicates, is-isync, is-sync and is-lwsync.

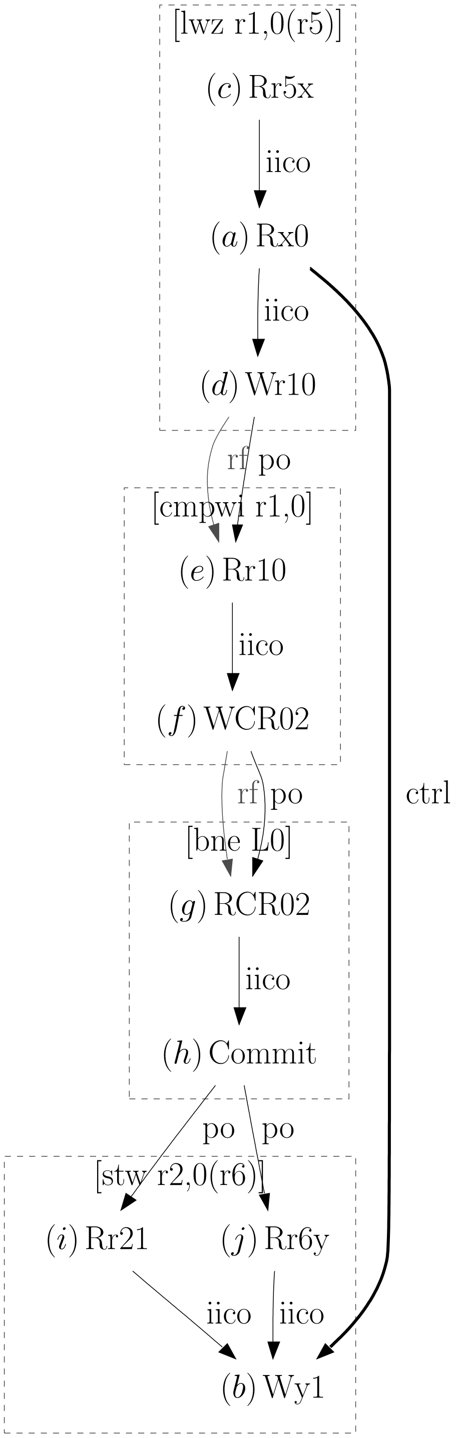


Figure 8.5: A Candidate Execution for the Test of Fig. 8.4

```
PPC isync
{
0:r5=x; 0:r6=y;0:r2=0;
x=0; y=1;
}
P0                ;
(1) lwz r1,0(r5) ;
(2) cmpwi r1,0    ;
(3) bne L0        ;
(4) isync         ;
(5) lwz r2,0(r6) ;
L0:              ;
exists(0:r2=1)
```

Figure 8.6: An Example Use of the `isync` Barrier

Consider for example the test given in Fig. 8.6, written in PowerPC assembly. Suppose that the register `r5` initially holds the address x, and the register `r6` the address y.

The `lwz r1,0(r5)` at line (1) and the `lwz r2,0(r6)` at line (5) are separated at line (2) by a compare instruction, written `cmpwi r1,0`, which influences the taking of the following branch at line (3), written `bne L0`. The branch is followed by an `isync` barrier at line (4).

In the execution of this test we give in Fig. 8.7, the `lwz r1,0(r5)` at line (1) leads to the read event $Rx0$ from x labelled (a). The `lwz r2,0(r6)` at line (5) leads to the read event $Ry1$ from y labelled (b). These two instructions are separated by a branch followed by an `isync`. This is depicted in the execution we give here by the commit event labelled (h), in $\xrightarrow{\text{po}}$ with the isync event labelled (i). Hence the read (a) from x and the read (b) from y are globally ordered by the `isync` barrier, as depicted by the $\xrightarrow{\text{isync}}$ arrow between them.

8.2.2 Description of the Model

We now give a description of our Power model. We give a full formalisation of this model in Fig. 8.8.

8.2.2.1 Preserved Program Order

We present in Fig. 8.8(a) the definition of $\xrightarrow{\text{ppo-ppc}}$, induced by lifting the ordering constraints of a processor to the global level (where $(\xrightarrow{\text{r}})^{+}$ stands for the transitive closure of a given relation $\xrightarrow{\text{r}}$). This is a formal presentation of the *data dependencies* ($\xrightarrow{\text{dd}}$) and *control dependencies* ($\xrightarrow{\text{ctrl}}$ and $\xrightarrow{\text{isync}}$)

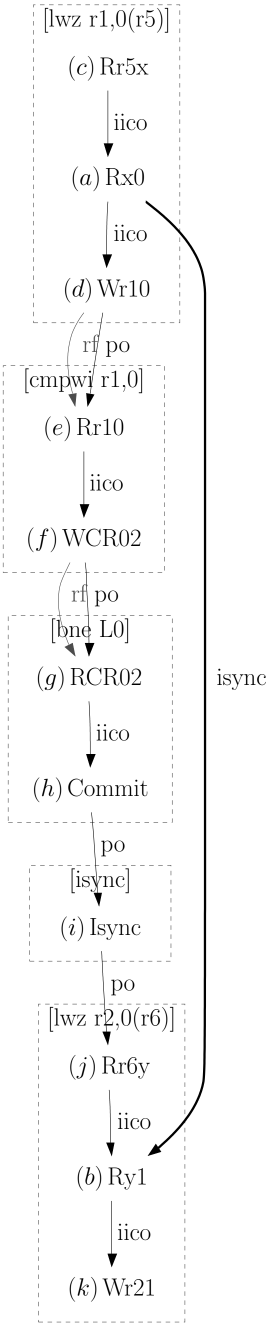


Figure 8.7: A Candidate Execution for the Test of Fig. 8.6

$$\xrightarrow{\text{dd}} \triangleq (\xrightarrow{\text{rf-reg}} \cup \xrightarrow{\text{iico}})^{+} \qquad\qquad r \xrightarrow{\text{ctrl}} w \triangleq \exists c \in \mathbb{C}.\ r \xrightarrow{\text{dd}} c \xrightarrow{\text{po}} w$$

$$r \xrightarrow{\text{isync}} e \triangleq \exists c \in \mathbb{C}.\ r \xrightarrow{\text{dd}} c \wedge \exists b.\ \text{is-isync}(b) \wedge c \xrightarrow{\text{po}} b \xrightarrow{\text{po}} e$$

$$\xrightarrow{\text{dp}} \triangleq \xrightarrow{\text{ctrl}} \cup \xrightarrow{\text{isync}} \cup \Big(\big(\xrightarrow{\text{dd}} \cup (\xrightarrow{\text{po-loc}} \cap (\mathbb{W} \times \mathbb{R}))\big)^{+} \cap (\mathbb{R} \times \mathbb{M})\Big) \qquad\qquad \xrightarrow{\text{ppo-ppc}} \triangleq \xrightarrow{\text{dp}}$$

(a) Preserved program order

$$\begin{array}{rcl}
m_1 \xrightarrow{\text{sync}} m_2 & \triangleq & \exists b.\ \text{is-sync}(b) \wedge m_1 \xrightarrow{\text{po}} b \xrightarrow{\text{po}} m_2 \\
m_1 \xrightarrow{\text{ab-sync}} m_2 & \triangleq & m_1 \xrightarrow{\text{sync}} m_2 \\
 & & \quad \vee\ \exists r.\ m_1 \xrightarrow{\text{rf}} r \xrightarrow{\text{ab-sync}} m_2 \\
 & & \quad \vee\ \exists w.\ m_1 \xrightarrow{\text{ab-sync}} w \xrightarrow{\text{rf}} m_2
\end{array}$$

(b) Barrier sync

$$\begin{array}{rcl}
m_1 \xrightarrow{\text{lwsync}} m_2 & \triangleq & \exists b.\ \text{is-lwsync}(b) \wedge m_1 \xrightarrow{\text{po}} b \xrightarrow{\text{po}} m_2 \\
m_1 \xrightarrow{\text{ab-lwsync}} m_2 & \triangleq & m_1 \xrightarrow{\text{lwsync}} m_2 \cap ((\mathbb{W} \times \mathbb{W}) \cup (\mathbb{R} \times \mathbb{M})) \\
 & & \quad \vee\ \exists r.\, m_1 \xrightarrow{\text{rf}} r \xrightarrow{\text{ab-lwsync}} m_2 \wedge m_2 \in \mathbb{W} \\
 & & \quad \vee\ \exists w.\, m_1 \xrightarrow{\text{ab-lwsync}} w \xrightarrow{\text{rf}} m_2 \wedge m_1 \in \mathbb{R}
\end{array}$$

(c) Barrier lwsync

$$\begin{array}{rcl}
\xrightarrow{\text{ab-ppc}} & \triangleq & \xrightarrow{\text{ab-sync}} \cup \xrightarrow{\text{ab-lwsync}} \\
\mathit{Power} & \triangleq & (\xrightarrow{\text{ppo-ppc}}, \emptyset, \xrightarrow{\text{ab-ppc}})
\end{array}$$

Figure 8.8: A Power Model

of [pow09, p. 661] which allow loads to be speculated if no isync is added after the branch but prevents stores from being speculated in any case.

Data Dependencies More precisely, we consider that two events are in data dependency, written $m_1 \overset{\text{dd}}{\to} m_2$, when:

- they are in $\overset{\text{rf-reg}}{\to}$, *i.e.* m_1 is a register write and m_2 a register read reading from m_1, or
- they are in $\overset{\text{iico}}{\to}$, *i.e.* they both come from the same instruction and the execution of m_2 depends on m_1, *e.g.* when using the value m_1 read, or
- there is a path of $\overset{\text{rf-reg}}{\to} \cup \overset{\text{iico}}{\to}$ between m_1 and m_2.

Control Dependencies We consider that two events are in control dependency, written $m_1 \overset{\text{ctrl}}{\to} m_2$, when:

- m_1 is a read and,
- m_2 is a write, and
- there exists a commit event c between them in the program order, such that c is in data dependency with m_1.

If two events m_1 and m_2 are in control dependency, it means that they form a read-write pair separated by a conditional jump or loop, and that the condition of the jump is data-dependent over m_1.

Moreover, it means that two events separated by a conditional jump may perfectly well be reordered if:

- the first one is a write, or
- they are both reads (*c.f.* infra, semantics of isync)

The test of Fig. 8.4 and the execution of Fig. 8.5 give an example of control dependency between the read (a) associated to the `lwz r1,0(r5)` at line (1) and the write (b) associated to the `stw r2,0(r6)` at line (3).

Semantics of the isync Barrier We now give our semantics of the isync barrier. The ordering induced by isync is similar to a control dependency on read-read pairs. We consider that two events m_1 and m_2 are ordered by an isync barrier when:

- m_1 is a read, and
- there exists a commit event c in data dependency with m_1, separated from m_2 by an isync barrier in the program order.

In particular, this means that two events m_1 and m_2 separated by an `isync` can be reordered, if:

- m_1 is a write, or
- there is no commit between m_1 and m_2.

The test of Fig. 8.6 and the execution of Fig. 8.7 give an example of `isync` ordering between the read (a) associated to `lwz r1,0(r5)` at line (1) and the read (b) associated to `lwz r2,0(r6)` at line (5).

All Together Finally, we consider that two events m_1 and m_2 are in Power's preserved program order, written $m_1 \overset{\text{ppo-ppc}}{\rightarrow} m_2$ when:

- they are in control dependency, *i.e.* $m_1 \overset{\text{ctrl}}{\rightarrow} m_2$, or
- they are ordered by an `isync` barrier, *i.e.* $m_1 \overset{\text{isync}}{\rightarrow} m_2$, or
- m_1 is a read, and there is a path of $\overset{\text{dd}}{\rightarrow} \cup (\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R}))$ between m_1 and m_2.

Resemblance to RMO's Preserved Program Order The preserved program order that we suggest for Power is similar to the one of Sparc RMO [spa94a, V9, p. 293]. Indeed, Sparc's documentation defines the *dependence order* as follows:

> *Dependence order is a partial order that captures the constraints that hold between instructions that access the same processor register or memory location. [...] Two memory transactions [i.e. accesses] X and Y are dependence ordered, denoted by $X <_d Y$, if and only if they are program ordered, [written] $X <_p Y$, **and** at least one of the following conditions is true:*
>
> *(1) The execution of Y is conditional on X, and $S(Y)$ is true. [i.e. they are in control dependency: in particular, Y is a write.]*
>
> *(2) Y reads a register that is written by X. [This corresponds to our $\overset{\text{rf-reg}}{\rightarrow}$ relation.]*
>
> *(3) X and Y access the same memory location, and $S(X)$ and $L(Y)$ are both true. [This corresponds to the $\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})$ part of Power's $\overset{\text{ppo}}{\rightarrow}$ definition.]*
>
> *[...] It is important to remember that partial ordering is transitive.*

The items (1) and (2) correspond to our $\xrightarrow{\text{ctrl}}$ and $\xrightarrow{\text{dd}}$ relations, while the item (3) corresponds to the $\xrightarrow{\text{po-loc}} \cap (\mathbb{W} \times \mathbb{R})$ part of Power's $\xrightarrow{\text{ppo}}$ definition. Moreover, as mentioned in the excerpt above, that order should be transitive. Hence we take the transitive closure of the relation formed by the union of these two relations.

Finally, we take the intersection of $(\xrightarrow{\text{dd}} \cup (\xrightarrow{\text{po-loc}} \cap (\mathbb{W} \times \mathbb{R})))^{+}$ and $\mathbb{R} \times \mathbb{W}$, exactly as in RMO [spa94a, V9, p.295], where the legality of a RMO memory order is defined as follows:

> *A memory order [written $<_m$] is legal [in RMO] if and only if:*
>
> *(1) $X <_d Y$ & $L(X) \Rightarrow X <_m Y$ [i.e. X and Y are in dependence order as defined above, and X is a read.]*
>
> *(2) $M(X,Y) \Rightarrow X <_m Y$ [i.e. X and Y are separated by a barrier; we treat this* via *$\xrightarrow{\text{ab}}$, not $\xrightarrow{\text{ppo}}$.]*
>
> *(3) $Xa <_p Ya$ & $S(Y) \Rightarrow X <_m Y$ [Y is a write, and they are both relative to the same memory location a. This corresponds to the fact that we consider $\xrightarrow{\text{wsi}}$ and $\xrightarrow{\text{fri}}$ to be global.]*

The first item indicates that RMO also only considers the dependency chains starting from a load, which explains why we take the intersection with $\mathbb{R} \times \mathbb{M}$.

Discussion of Power's $\xrightarrow{\text{ppo}}$ We include in $\xrightarrow{\text{ppo}}$ any chain of data dependencies and $\xrightarrow{\text{rfi}}$ starting from a read, by the $((\xrightarrow{\text{dd}} \cup (\xrightarrow{\text{po-loc}} \cap (\mathbb{W} \times \mathbb{R})))^{+} \cap (\mathbb{R} \times \mathbb{M})$ part. However, we do not authorise control dependencies in such a chain. Indeed, consider the example given in Fig. 8.9, which we observed to be exhibited on a Power 5 for example.

On P_0, the `lwz r1,0(r7)` at line (1) is in control dependency with the `stw r3,0(r9)` at line (5), because of the compare and branch sequence between them (lines (2) to (4)). The `lwz r2,0(r9)` at line (6) is in data dependency with the `lwzx r4,r10,r8` at line (8), because of the `xor r10,r2,r2` between them, at line (7).

Since the `lwz r1,0(r7)` at line (1) on P_0 reads 1 (see the final state, `0:r1=1`), there is a $\xrightarrow{\text{rfe}}$ between the `stw r2,0(r7)` at line (3) on P_1 and the `lwz r1,0(r7)` at line (1) on P_0. Because of the A-cumulativity of the `sync` barrier on P_1, the `stw r1,0(r8)` at line (1) on P_1 is in $\xrightarrow{\text{ab}}$ with the `lwz r1,0(r7)` at line (1) on P_0.

Since the `lwzx r4,r10,r8` at line (8) on P_0 reads 0 (see the final state, `0:r4=0`), there is a $\xrightarrow{\text{fr}}$ between this load and the `stw r1,0(r8)` at line (1) on P_1.

```
{
0:r7=y; 0:r8=z; 0:r9=x; 0:r3=1;
1:r7=y; 1:r8=z; 1:r1=1; 1:r2=1;
}
P0                   | P1             ;
(1) lwz r1,0(r7)     | stw r1,0(r8) ;
(2) cmpwi r1,0       | sync           ;
(3) beq L0           | stw r2,0(r7) ;
(4) L0:              |                ;
(5) stw r3,0(r9)     |                ;
(6) lwz r2,0(r9)     |                ;
(7) xor r10,r2,r2    |                ;
(8) lwzx r4,r10,r8   |                ;
exists (0:r1=1  /\ 0:r4=0)
```

Figure 8.9: Contradiction with the View Order Formulation

Hence we have a cycle of relations as follows, starting from `lwz r1,0(r7)` at line (1) on P_0 (writing $(i : j)$ for the instruction at line (j) on P_i):

$$(0:1) \xrightarrow{\text{CtrldW}} (0:5) \xrightarrow{\text{PosWR}} (0:6) \xrightarrow{\text{DpdR}} (0:8) \xrightarrow{\text{Fre}} (1:1) \xrightarrow{\text{ACSyncdWW}} (0:1)$$

Since the specified outcome is exhibited, we know that there is a subsequence of this cycle which is not global. The only possible relaxation here is the PosWR one between the `stw r3,0(r9)` at line (5) and the `lwz r2,0(r9)` at line (6). If we added $\overset{\text{ctrl}}{\to}$ to the transitive part of Power's $\overset{\text{ppo}}{\to}$, the outcome would be forbidden in our model. We deduce from this example that $\overset{\text{ctrl}}{\to}$ cannot be included in such chains.

8.2.2.2 Read-From Maps

Internal Read-From Maps The internal read-from maps are not global, since Power allows store buffering (see Sec. 6.1.2). Running the test given in Fig. 8.10 confirms this hypothesis. Indeed this test, generated by diy, proceeds from a cycle $\xrightarrow{\text{DpdR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{Rfi}};\xrightarrow{\text{DpdR}};\xrightarrow{\text{Fre}};\xrightarrow{\text{Rfi}}$. We know that $\overset{\text{dp}}{\to}$ is global, since included in $\overset{\text{ppo}}{\to}$, and $\overset{\text{fr}}{\to}$ is global as well. Thus in this test, the only possible relaxation is $\overset{\text{rfi}}{\to}$. Since the outcome is exhibited, as shown in Fig. 8.1, we know that $\overset{\text{rfi}}{\to}$ is actually relaxed on Power.

Store buffering is a fairly common relaxation. Indeed, all the models we presented in Chap. 4, except SC, also relax their internal read-from maps, $i.e.$ allow store buffering.

External Read-From Maps The external read-from maps are not global either, as revealed by running **iriw** with data dependencies (Fig. 7.5) on a

```
PPC rfi000
"DpdR Fre Rfi DpdR Fre Rfi"
Cycle=DpdR Fre Rfi DpdR Fre Rfi
Relax=Rfi
Safe=Fre DpdR
{
0:r2=x; 0:r6=y;
1:r2=y; 1:r6=x;
}
 P0             | P1             ;
 li r1,1        | li r1,1        ;
 stw r1,0(r2)   | stw r1,0(r2)   ;
 lwz r3,0(r2)   | lwz r3,0(r2)   ;
 xor r4,r3,r3   | xor r4,r3,r3   ;
 lwzx r5,r4,r6  | lwzx r5,r4,r6  ;
exists
(0:r3=1 /\ 0:r5=0 /\ 1:r3=1 /\ 1:r5=0)
```

Figure 8.10: A Diy PowerPC Test for the Rfi Relaxation

Power machine. Indeed, this test is associated to a cycle $\xrightarrow{\text{Fre}}; \xrightarrow{\text{Rfe}}; \xrightarrow{\text{DpdR}}; \xrightarrow{\text{Fre}}$ $; \xrightarrow{\text{Rfe}}; \xrightarrow{\text{DpdR}}$. We know that $\overset{\text{fre}}{\to}$ is always considered global in our framework, an hypothesis which has not been invalidated by the experiment we present in Sec. 8.1. We also know that $\overset{\text{dp}}{\to}$ is global in Power, since $\overset{\text{ppo}}{\to}$ is equal to $\overset{\text{dp}}{\to}$. Therefore, the only possible relaxation in the test of Fig. 7.5 is $\overset{\text{rfe}}{\to}$. Since the specified outcome is exhibited as shown in Fig. 8.1, $\overset{\text{rfe}}{\to}$ is relaxed.

This is probably the main particularity of the Power architecture: all the models we presented in Chap. 4 do not relax the atomicity of stores. Another model relaxing store atomicity may be Itanium [ita02]. However, we do not know whether Itanium could be described by the generic framework we presented in Chap. 3.

8.2.2.3 Barriers

The sync Barrier The sync barrier is defined in Fig. 8.8 (b) as a full A- and B-cumulative barrier. We will see in Sec. 10.3.3.1 that such a barrier restores *SC* from a weaker model.

The lwsync Barrier The lwsync barrier is defined in Fig. 8.8 (b). lwsync acts as sync except on store-load pairs, in both the base and cumulativity cases.

P_0	P_1
(a) x ← 1	(d) r1 ← p
(b) p ← & x	(c) r2 ← *r1
Observed? r1=& x; r2=0;	

Figure 8.11: A Common Programming Idiom

```
{0:r5=x; 0:r6=y; 0:r3=1; 0:r4=1;
 1:r5=x; 1:r6=y; 1:r1=0; 1:r9=0; 1:r2=0;
 x=0; y=0;}

P0            | P1                ;
stw r3,0(r5)  | lwz r1,0(r6)      ;
lwsync        | xor r9,r1,r1      ;
stw r4,0(r6)  | lwzx r2,r9,r5     ;

exists(1:r3=1 /\ 1:r4=0)
```

Figure 8.12: The Example of Fig. 8.11 in PowerPC Assembly

8.3 Discussion of Our Model

We do not claim our model to be definitive, as we explain in the following. In particular, our model is probably too coarse-grained. To refine our model, we would probably need to design a less coarse-grained model, in which either we would split the write events into several ones, in the spirit of Itanium [ita02], or in which we would examine the views of each processor separately, instead of considering a single global time line. Indeed the documentation is in terms of view orders, since it specifies the events to be "performed *w.r.t.*" a processor, or all processors.

For example, consider the common programming idiom given in Fig. 8.11 in C-like syntax. We are unable to explain why this example is guaranteed not to happen if the two instructions on P_0 are globally maintained *e.g.* by a `lwsync` barrier, because our semantics for the `lwsync` barrier is rather weak.

Consider indeed the PowerPC example we give in Fig. 8.12, which corresponds to the idiom of Fig. 8.11. The example given in Fig. 8.12 could, according to our Power model, exhibit the execution given in Fig. 8.13 since there is no cycle in $\xrightarrow{\text{ghb}}$. Hence this behaviour is not forbidden by our model, though it is expected to be.

A per-processor view of the memory helps to explain why the example of Fig. 8.13 is guaranteed not to happen. Indeed, the definition of `lwsync` of the documentation (see Sec. 6.1.5.1) specifies (since the pair $((a), (b))$ is a write-

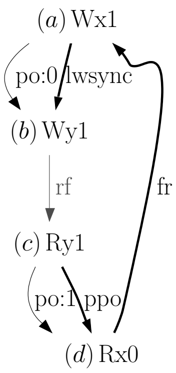


Figure 8.13: Weakness of Our `lwsync` Semantics

write pair, hence applicable to `lwsync`) that "the memory barrier ensures that $[(a)]$ will be performed with respect to any processor $[\ldots]$ before $[(b)]$ is performed with respect to that processor $[\ldots]$." Hence the pair $((a),(b))$ has to be seen in the same order by P_0 and P_1, because there is a `lwsync` between (a) and (b). Therefore, if the read (c) on P_1 reads 1 from the write (b) on P_0, then P_1 has also seen the write (a) to x on P_0, hence we cannot have $(d) \xrightarrow{\text{fr}} (a)$ as in Fig. 8.13.

We would like to reconcile our model and the one that is presented in the Power documentation, in the light of what we learnt during our experiment. However, even though our model should be further refined, this is a first non-trivial attempt towards the formalisation of the Power architecture. We consider this attempt as a success because the model, though a bit loose, has not been invalidated by our thorough experiment: at least it is a valid model. Moreover, our testing methodology can perfectly well be reused, and adapted if needed to the study of other architectures, as we sketched with the x86 example of Sec. 7.4, or to investigate further the Power architecture.

Chapter 9

Related Work

We present here some related work, either using the concept of relaxation or focussing on the testing of weak memory models. Finally, we present some studies of the Power architecture.

The framework we presented in Chap. 3 owes much to the concept of relaxation, as used in the documentations we consider formal (*i.e.* Alpha's and Sparc's), and presented in [AG95].

More generally, several papers on the design of abstract memory models use a similar concept, *e.g.* W. Collier's [Col92]. Moreover, it seems to me that the concept of relaxation is really well illustrated by litmus tests, as we tried to explain in this part of the manuscript. Hence, the Intel White Paper on memory ordering [int07] provides an informal memory model of the x86 architecture in terms of ten litmus tests, each of which illustrates one or several relaxations. Several other papers on memory models, both on hardware and software, illustrate their formalisation with such tests, *e.g.* [BA, ASb].

W. Collier's framework is, like ours, used to do model exploration *via* litmus tests—like we did with our experiment on the Power architecture. However, W. Collier's framework does not provide a way to generate tests systematically as we do.

A. Adir *et al.* 's model [AAS03] also provides a set of litmus tests to illustrate their formalisation of the Power architecture. However, their work does not include a systematic test generation, and, like W. Collier's, does not adress the cumulativity of barriers.

A recent work by S. Mador-Haim *et al.* describes a method for systematically generating litmus tests [MHAM]. However, their tests serve a different purpose. The goal is to distinguish two models by a given test: if the outcome of a test is observable on one model and not on the other, the two models are distinct. This may be used for example to highlight specification bugs: if the specification of an architecture can be distinguished from a machine with the same architecture, then either the specification is false, or the machine has a bug. This can be related to the implementation error we found on Power 5: the **iriw** example distinguishes our Power model from the Power 5 implementation we tested.

Part IV

Synchronisation in Weak Memory Models

We know that an execution of a program running on a multiprocessor may not be an interleaving of its instructions, as it would be on a Sequentially Consistent (SC) architecture [Lam79]. Fortunately, a program running on a weak architecture may be constrained to behave as if it were running on a stronger one (*e.g.* SC) by *synchronisation primitives*.

Two approaches coexist: *lock-based* and *lock-free* synchronisation. Both ensure *stability* of a program: the program has no other executions than the ones valid on SC. Locks allow the programmer to ignore the details of the memory model, thanks to the *data race free guarantee* (DRF guarantee) proposed by S. Adve and M. Hill [AH], but are costly. On the other hand, lock-free techniques are efficient [Her], but require a detailed knowledge of the memory model. For instance, D. Shasha and M. Snir proposed the *delay set analysis* to ensure stability to a program by inserting *barriers* [SS].

We examine here how to force all the executions of a program running on a weak architecture A_1 to be valid on a stronger one A_2, *i.e.* we examine when the following property holds for all executions (E, X) of a program (writing $A.\,\mathsf{valid}(E, X)$ when (E, X) is valid on the architecture A):

$$\mathsf{stable}_{\mathsf{A_1,A_2}}(E, X) \triangleq (A_1.\,\mathsf{valid}(E, X) \Leftrightarrow A_2.\,\mathsf{valid}(E, X))$$

A program that is not stable is often considered as erroneous: for example, a program with data races may have non-SC behaviours. Hence, checking the stability of a program is a crucial step towards checking its correctness. Moreover, a precise study of the stability of a program can lead to optimisations, as we illustrate here. Indeed, the stability of a program can be naively enforced by placing synchronisation primitives everywhere; but using too much synchronisation can severely impair the performance of this program. Showing exactly where a program needs to be synchronised, *i.e.* how to make it stable, can help improve its performance.

Ideally, we want to examine the stability of a program by checking its executions on the strong architecture A_2 only, because the executions on A_2 are easier to reason about than on A_1, since they endure less relaxations. As a consequence, the design of a proof or an algorithm checking this property will be easier.

We provide in Sec. 10.1 a generic sufficient condition over the strong executions (*i.e.* the ones valid on A_2) ensuring stability. To do so, we describe a class of synchronisation relations, the *covering* and *well-founded* relations. We show that if all the *conflicting* accesses of a given execution X valid on A_2 are *arbitrated* by such a relation, then X is stable.

Hence, we propose a unifying approach to describe both lock-based and lock-free synchronisation. We instantiate this approach to produce a generalisation of the DRF guarantee in Sec. 10.2 and a novel dual result on barrier placement in Sec. 10.3, which we call the *lock-free guarantee*.

We illustrate these two guarantees by the study of Power's lock and *read-modify-write* primitives (*e.g.* *test-and-set* and *compare-and-swap*): we propose in Sec. 10.4 a semantics for these primitives. We show that locks ensure the DRF guarantee, and that mapping certain loads to read-modify-write primitives coupled with non-cumulative barriers restores SC.

Moreover, we refine the data race free and lock-free guarantees in the style of [SS] in Sec. 11.1, in order to minimise the required amount of synchronisation. Finally, we show in Sec. 11.2 that it is necessary and sufficient to synchronise the *minimal cycles* of an execution valid on SC to ensure its stability. This gives a characterisation of the stable executions.

In the following, we consider the architecture A_2 to be without barriers, *i.e.* $\overset{\mathrm{ab}_2}{\rightarrow} = \emptyset$. We write $\overset{\mathrm{ghb}_2}{\rightarrow}$ for $A_2 . \mathsf{ghb}(E, X)$. Moreover, although the definition of $\mathsf{stable}_{\mathsf{A}_1,\mathsf{A}_2}$ holds for any pair of architectures, we focus here on two architectures A_1 and A_2 such that $A_1 \leq A_2$.

Chapter 10

Synchronisation

Synchronisation idioms are used to *arbitrate conflicts* between accesses, *i.e.* to ensure that one out of two conflicting accesses occurs before the other.

10.1 Covering and well-founded relations

To formalise this notion, given an execution (E, X), we postulate an irreflexive *conflict* relation $\xrightarrow{\text{c}}$, such that $\forall xy, x \xrightarrow{\text{c}} y \Rightarrow \neg(y \xrightarrow{\text{po}} x)$ and a *synchronisation* relation $\xrightarrow{\text{s}}$ over its events. An execution (E, X) is *covered* when the pairs of $\xrightarrow{\text{c}}$ are *arbitrated* by $\xrightarrow{\text{s}}$, which means that the conflicts gathered in $\xrightarrow{\text{c}}$ are ordered by $\xrightarrow{\text{s}}$:

Definition 48 (Covered Execution)

$$\mathsf{covered}(E, X, \xrightarrow{\text{c}}, \xrightarrow{\text{s}}) \triangleq \forall xy, x \xrightarrow{\text{c}} y \Rightarrow x \xrightarrow{\text{s}} y \vee y \xrightarrow{\text{s}} x$$

We define in the following a class of synchronisation relations which we show to ensure stability. We then illustrate two example synchronisations of this class, which lead us to a proof that a generalised DRF guarantee holds for the framework presented in Chap. 3, and a novel dual result on lock-free synchronisation, which we call the lock-free guarantee.

10.1.1 Covering relations

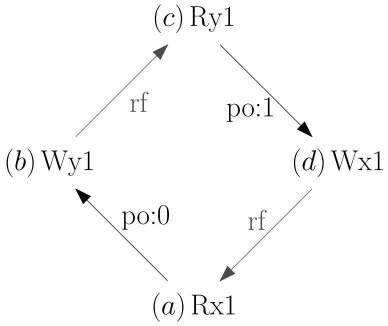
We consider a relation $\xrightarrow{\text{s}}$ to be *covering* when ordering by $\xrightarrow{\text{s}}$ the conflicting accesses of an execution (E, X) valid on A_1 guarantees the validity of (E, X) on A_2. This means that the chosen synchronisation $\xrightarrow{\text{s}}$ arbitrates indeed enough conflicts in (E, X) to enforce a strong behaviour. We know by Thm. 2 that given an execution valid on A_1 it is enough to enforce the acyclicity of $\xrightarrow{\text{ghb}_2}$ to ensure the validity of the execution on A_2, hence we formalise the notion of covering as follows:

Init: `x=0; y=0; z=0`

P_0	P_1
(a) `r1` $\leftarrow$ `x`	(c) `r2` $\leftarrow$ `y`
(b) `y` $\leftarrow$ `1`	(d) `x` $\leftarrow$ `1`

Observed? `r1=1; r2=1;`

(a) A program



(b) A non-SC execution

Figure 10.1: A program and a non-SC execution

Definition 49 (Covering Relation)

$$\begin{aligned}\mathsf{covering}(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) \triangleq \forall EX, (&A_1 \,.\, \mathsf{valid}(E,X) \wedge \\ &\mathsf{covered}(E, X, \xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}})) \Rightarrow \mathsf{acyclic}(A_2 \,.\, \mathsf{ghb}(E,X))\end{aligned}$$

For example, the DRF guarantee [AH] stipulates in particular that if certain accesses (the *competing accesses*, defined in Sec. 10.2) of a given execution are ordered by a given synchronisation relation $\xrightarrow{\mathrm{sync}}$, then this execution is SC. This means that $\xrightarrow{\mathrm{sync}}$ is covering *w.r.t.* the competing accesses.

10.1.2 Well-founded relations

We consider a synchronisation relation to be *well founded* when each conflict it arbitrates on the strong architecture is arbitrated on the weaker one. This means that it is enough to arbitrate the conflicts on the strong executions of a program to ensure that every execution of this program will be strong, even if it is running on a weak architecture.

Formally, we define a synchronisation relation to be well founded when, given an uncovered execution (E, X) valid on A_1, one can build an uncovered execution Y associated with the same event structure E and valid on A_2:

Definition 50 (Well-Founded Synchronisation)

$$\begin{aligned}\mathsf{wf}(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) \triangleq{}& \forall EX, \\ &\Big(A_1 \,.\, \mathsf{valid}(E,X) \wedge \neg(\mathsf{covered}(E, X, \xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}))\Big) \Rightarrow \\ &\Big(\exists Y, A_2 \,.\, \mathsf{valid}(E,Y) \wedge \neg(\mathsf{covered}(E, Y, \xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}))\Big)\end{aligned}$$

This means that given a conflicting pair (x, y) unarbitrated by $\xrightarrow{\mathrm{s}}$ in an execution (E, X) on a weak architecture A_1, if (x, y) is also conflicting in an execution (E, Y) on a stronger architecture A_2, then (x, y) is unarbitrated by $\xrightarrow{\text{s}}$ in (E, Y). In other words, this means that the conflicting accesses to arbitrate stay the same from one architecture to another. This will allow us to check these conflicting accesses on the strong executions only.

For example, the competing accesses of the DRF guarantee are accesses to the same location, from distinct processors, such that one of them at least is a write. If we have an execution (E, X) where two such competing accesses are not arbitrated by a lock for example, then we can build a SC execution containing the same accesses, where they will not be arbitrated by a lock either. This means that the synchronisation relation induced by the locks is well-founded $w.r.t.$ the competing accesses.

Finally, we show that if a well-founded synchronisation relation covers any execution valid on A_2 then any execution is stable:

Theorem 8 (Well-founded and covering ensure stability)

$$\begin{array}{l} \forall \xrightarrow{\text{c}}\xrightarrow{\text{s}}, \mathsf{wf}(\xrightarrow{\text{c}}, \xrightarrow{\text{s}}) \wedge \mathsf{covering}(\xrightarrow{\text{c}}, \xrightarrow{\text{s}}) \Rightarrow \\ \quad ((\forall EX, A_2 \,.\, \mathsf{valid}(E, X) \Rightarrow \mathsf{covered}(E, X)) \Rightarrow \\ \quad\quad (\forall EX, \mathsf{stable}_{\mathsf{A_1,A_2,(}}\, E, X))) \end{array}$$

Proof *Let E be an event structure; suppose that any associated execution (E, X) valid on A_2 is covered. Let us show that in this case, any execution (E, X) is stable, i.e. $A_1 \,.\, \mathsf{valid}(E, X) \Leftrightarrow A_2 \,.\, \mathsf{valid}(E, X)$. The converse way $A_2 \,.\, \mathsf{valid}(E, X) \Rightarrow A_1 \,.\, \mathsf{valid}(E, X)$ is given by Thm. 1, since A_1 is weaker than A_2.*

Conversely, suppose (E, X) valid on A_1. If (E, X) is covered, since $\xrightarrow{\text{s}}$ is covering, we know by definition that (E, X) is valid on A_2. If (E, X) is uncovered, since $\xrightarrow{\text{s}}$ is well-founded, we know by definition that there exists an execution Y associated with E, valid on A_2 and uncovered. But this contradicts our hypothesis that any execution associated with E and valid on A_2 is covered. □

This result means that, provided a well-founded and covering synchronisation relation, it is enough to arbitrate the conflicts occurring in all the strong executions in order to ensure stability. In other words, this means that we only have to reason on the strong executions of a program to ensure that this program behaves as if it were running on a strong architecture, typically SC.

10.2 DRF guarantee

We propose here a first example of covering relation, leading to a proof that a generalised DRF guarantee holds for each instance of the framework of Chap. 3.

10.2.1 Competing accesses

We follow the DRF$_0$ model of [AH], where two events are *competing* if they are from two distinct processors, relative to the same location, and one of them at least is a write:

Definition 51 (Competing Accesses)

$$\begin{aligned} m_1 \overset{\mathrm{c_{drf}}}{\leftrightarrow} m_2 \quad &\triangleq \quad \mathsf{proc}(m_1) \neq \mathsf{proc}(m_2) \wedge \mathsf{loc}(m_1) = \mathsf{loc}(m_2) \wedge \\ &\quad\; (m_1 \in \mathbb{W} \vee m_2 \in \mathbb{W}) \end{aligned}$$

Consider the program given in Fig. 10.1(a): the read (a) from x on P_0 competes with the write (d) to x on P_1. Similarly, the write (b) to y on P_0 and the read (c) from y on P_1 are competing.

Synchronisation We postulate a relation $\xrightarrow{\mathrm{sync}}$ over $\mathbb{E}$ compatible with $\xrightarrow{\mathrm{com}}$, and define our synchronisation relation as the transitive closure of $\xrightarrow{\mathrm{sync}} \cup \xrightarrow{\mathrm{po}}$, *i.e.* the *happens-before* of [AH]:

Definition 52 (Lock-Based Synchronisation)

$$\xrightarrow{\mathrm{s_{drf}}} \;\triangleq\; \left(\xrightarrow{\mathrm{sync}} \cup \xrightarrow{\mathrm{po}}\right)^{+}$$

This synchronisation relation corresponds for example to the order induced by locks on competing accesses. For example in Fig. 10.1(a), this corresponds to placing locks to a variable ℓ_1 on the accesses (a) and (d) relative to x, in which case we have $(d) \xrightarrow{\mathrm{sync}} (a)$, and locks to a different variable ℓ_2 on the accesses (b) and (c) relative to y, *i.e.* $(b) \xrightarrow{\mathrm{sync}} (c)$.

Finally, an execution is covered in the DRF sense if all the competing pairs are arbitrated by $\xrightarrow{\mathrm{s_{drf}}}$:

Definition 53 (DRF Covered)

$$\mathsf{covered}_{\mathrm{DRF}}(X) \;\triangleq\; \forall xy, x \overset{\mathrm{c_{drf}}}{\leftrightarrow} y \Rightarrow x \xrightarrow{\mathrm{s_{drf}}} y \vee y \xrightarrow{\mathrm{s_{drf}}} x$$

10.2.2 Synchronising competing accesses in a weak execution

Consider the example in Fig. 10.1(a), and suppose that we have placed locks to ℓ_1 on the accesses (a) and (d) relative to x (*i.e.* $(d) \xrightarrow{\mathrm{sync}} (a)$), and locks to ℓ_2 on the accesses (b) and (c) relative to y (*i.e.* $(b) \xrightarrow{\mathrm{sync}} (c)$). Hence we have a cycle in $\xrightarrow{\mathrm{sync}} \cup \xrightarrow{\mathrm{po}}$: $(a) \xrightarrow{\mathrm{po}} (b) \xrightarrow{\mathrm{sync}} (c) \xrightarrow{\mathrm{po}} (d) \xrightarrow{\mathrm{sync}} (a)$. If we have the guarantee that $\xrightarrow{\mathrm{sync}}$ is compatible with $\xrightarrow{\mathrm{po}}$ (*i.e.* $\xrightarrow{\mathrm{sync}} \cup \xrightarrow{\mathrm{po}}$ is acyclic), then this cycle is forbidden, which forbids the execution of Fig. 10.1(b).

Indeed, we show that if $\xrightarrow{\mathrm{s_{drf}}}$ is irreflexive (*i.e.* $\xrightarrow{\mathrm{sync}} \cup \xrightarrow{\mathrm{po}}$ acyclic), then $\xrightarrow{\mathrm{s_{drf}}}$ is covering:

Lemma 10 (An irreflexive $\xrightarrow{\mathrm{s_{drf}}}$ is covering)

$$\mathsf{irreflexive}(\xrightarrow{\mathrm{s_{drf}}}) \Rightarrow \mathsf{covering}(\overset{\mathrm{c_{drf}}}{\leftrightarrow}, \xrightarrow{\mathrm{s_{drf}}})$$

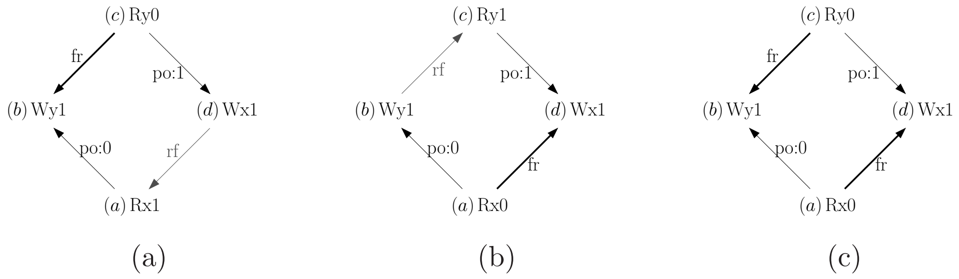


Figure 10.2: SC executions for the test of Fig. 10.1(a)

Proof *Let (E, X) be valid on A_1 and covered. Suppose by contradiction that there is a cycle in $\xrightarrow{\text{ghb}_2}$, which is by definition a cycle in $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{grf}_2} \cup \xrightarrow{\text{ppo}_2}$. The events in $\xrightarrow{\text{ws}}$, $\xrightarrow{\text{fr}}$ or $\xrightarrow{\text{rf}}$ and from distinct processors are competing. Since (E, X) is covered, $\xrightarrow{\text{sync}}$ orders the competing accesses according to $\xrightarrow{\text{com}}$. The remaining events in $\xrightarrow{\text{com}}$ belong to the same processor, hence are in $\xrightarrow{\text{po}}$ by* uniproc. *Moreover, we know $\xrightarrow{\text{ppo}_2} \subseteq \xrightarrow{\text{po}}$. Hence a cycle in $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{grf}_2} \cup \xrightarrow{\text{ppo}_2}$ is a cycle in $\xrightarrow{\text{sync}} \cup \xrightarrow{\text{po}}$, which contradicts the irreflexivity of $\xrightarrow{\text{s}_{\text{drf}}}$.* □

This means that if we synchronise the competing accesses of a weak execution ($e.g.$ in Fig. 10.1(b)), (a) and (d) on the one hand and (b) and (c) on the other hand) then this execution is either valid on A_2, or forbidden on A_1.

10.2.3 DRF guarantee

We now refine this result: we show that it is enough to synchronise the competing accesses of all the strong executions to ensure that there cannot be any weak execution.

Indeed, the DRF guarantee of [AH] stipulates that if the competing accesses ($i.e.$ $\overset{\text{c}_{\text{drf}}}{\leftrightarrow}$) of the SC executions of a given program are arbitrated by $(\xrightarrow{\text{sync}} \cup \xrightarrow{\text{po}})^+$ ($i.e.$ by $\xrightarrow{\text{s}_{\text{drf}}}$), then there can be no other executions. We generalise this to any $A_1 \leq A_2$:

Definition 54 (DRF **Guarantee from A_1 to A_2)**

$$\begin{array}{l} A_1 \,.\, \mathrm{drfg}(A_2) \triangleq \\ \quad (\forall EX, A_2 \,.\, \mathsf{valid}(E, X) \Rightarrow \mathsf{covered}_{\mathrm{DRF}}(E, X)) \Rightarrow \\ \qquad\qquad (\forall EX, \mathsf{stable}_{\mathsf{A_1,A_2,(}}\ E, X)) \end{array}$$

We showed in Lem. 10 that $\xrightarrow{\text{s}_{\text{drf}}}$ is covering if it is irreflexive. Hence Thm. 8 ensures that, if $\xrightarrow{\text{s}_{\text{drf}}}$ is irreflexive and well-founded, our generalised DRF guarantee holds:

Corollary 6 (Irreflexivity of $\xrightarrow{\text{s}_{\text{drf}}}$ ensures stability)

$$\forall A_1 \leq A_2, \mathsf{wf}(\overset{\text{c}_{\text{drf}}}{\leftrightarrow}, \xrightarrow{\text{s}_{\text{drf}}}) \land \mathsf{irreflexive}(\xrightarrow{\text{s}_{\text{drf}}}) \Rightarrow A_1 \,.\, \mathrm{drfg}(A_2)$$

This means that we only have to synchronise the competing accesses of all the strong executions of a program to ensure its stability. Consider again the test given in Fig. 10.1(a). We give in Fig. 10.2 the additional executions of this test. They are valid on SC since they do not exhibit any cycle in $SC\,.\,\mathsf{ghb}$ ($i.e.$ in $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$). Cor. 6 shows that it is enough to arbitrate the competing accesses of the SC executions of Fig. 10.2 to prevent the non-SC execution of Fig. 10.1(b) from happening. Hence in Fig. 10.2, we have to order (a) and (d) on the one hand and (b) and (c) on the other hand. The pairs to be arbitrated are the same as in Fig. 10.1(b). However, we only had to examine the SC executions to ensure that there can be no other execution but those ones.

10.3 Lock-free guarantee

We give here another example of covering relation, relative to lock-free synchronisation.

10.3.1 Fragile pairs

We saw in Chap. 3 that two architectures $A_1 \leq A_2$ can be distinguished according to their preserved program order ($i.e.$ which pairs in $\xrightarrow{\text{po}}$ they maintain in this order for every processor involved), and their policy $w.r.t.$ store atomicity ($i.e.$ which writes are considered to be committed to memory as soon as they are issued). Hence, a same program can exhibit different executions on A_1 and A_2 for two reasons.

First, if the program involves a pair (x, y) of events that are maintained in program order on A_2 ($i.e.$ $x \xrightarrow{\text{ppo}_2} y$) but not on A_1 ($i.e.$ $\neg(x \xrightarrow{\text{ppo}_1} y)$). For example in Fig. 10.1(b), the read (a) from x on P_0 is in program order with the write (b) to y on P_0, $i.e.$ $(a) \xrightarrow{\text{po}} (b)$. Hence on a strong architecture A_2 such as SC where $\xrightarrow{\text{ppo}_2}=\xrightarrow{\text{po}}$, we have $(a) \xrightarrow{\text{ppo}_2} (b)$. On a weak architecture A_1 such as Power, where the read-write pairs in program order are not maintained, we have $\neg((a) \xrightarrow{\text{ppo}_1} (b))$.

Second, if the program involves a read reading from a write that is considered atomic on A_2 but not on A_1. For example in Fig. 10.1(b), the read (a) from x on P_0 reads from the write (d) to x on P_1, *via* an external read-from map (since (a) and (d) belong to different processors). Hence we have $(d) \xrightarrow{\text{rfe}} (a)$. On a strong architecture A_2 such as SC where the writes are atomic, $i.e.$ $\xrightarrow{\text{grf}}=\xrightarrow{\text{rf}}$, we have $(d) \xrightarrow{\text{grf}} (a)$. On a weak architecture A_1 such as Power, which relaxes store atomicity, we have $\neg((d) \xrightarrow{\text{grf}} (a))$

We consider such differences between architectures as conflicts, and formalise this notion as follows. We consider that two events form a *fragile pair* if they are maintained in the program order on A_2, and either they are not maintained in the program order on A_1, or the first event reads from a write that is atomic on A_2 but not on A_1. We call such reads *fragile reads* and define them as follows ($\xrightarrow{\mathrm{r}_{2\setminus 1}} \triangleq \xrightarrow{\mathrm{r}_2} \setminus \xrightarrow{\mathrm{r}_1}$ being the set difference):

Definition 55 (Fragile Read)

$$\mathsf{fragile}(r) \quad \triangleq \quad \exists w, w \xrightarrow{\mathrm{grf}_{2\setminus 1}} r$$

Finally, we define the conflicting pairs $\xrightarrow{\mathrm{c}_{\mathrm{lf}}}$ as the fragile pairs:

Definition 56 (Fragile Pairs)

$$m_1 \xrightarrow{\mathrm{c}_{\mathrm{lf}}} m_2 \quad \triangleq \quad m_1 \xrightarrow{\mathrm{ppo}_2} m_2 \wedge \big(\neg(m_1 \xrightarrow{\mathrm{ppo}_1} m_2) \vee \mathsf{fragile}(m_1)\big)$$

For example, in Fig. 10.1(b), the read (a) from x on P_0 is fragile on an architecture A_1 relaxing store atomicity, since it reads from the write (d) to x on P_1 *via* a $\xrightarrow{\mathrm{rf}}$ that is not global. Suppose that A_2 preserves the read-write pairs: in this case the events (a) and (b) form a fragile pair on P_0, since they are in $\xrightarrow{\mathrm{ppo}_2}$ and (a) is a fragile read. Similarly, the events (c) and (d) on P_1 form a fragile pair.

10.3.2 Synchronising fragile pairs in a weak execution

An execution is covered if the fragile pairs are arbitrated by a synchronisation relation $\xrightarrow{\mathrm{s}_{\mathrm{lf}}}$:

Definition 57 (LF Covered)

$$\mathsf{covered}_{\mathrm{LF}}(X) \ \triangleq \ \forall xy, x \xrightarrow{\mathrm{c}_{\mathrm{lf}}} y \Rightarrow x \xrightarrow{\mathrm{s}_{\mathrm{lf}}} y \vee y \xrightarrow{\mathrm{s}_{\mathrm{lf}}} x$$

This synchronisation relation corresponds intuitively to the order induced by placing barriers between the events of the fragile pairs. For example in Fig. 10.1(b), this corresponds to placing a barrier between (c) and (d) on P_1, in which case we have $(c) \xrightarrow{\mathrm{s}_{\mathrm{lf}}} (d)$, and another barrier between (a) and (b) on P_0, in which case we have $(a) \xrightarrow{\mathrm{s}_{\mathrm{lf}}} (b)$. Hence we have a cycle in $\xrightarrow{\mathrm{s}_{\mathrm{lf}}} \cup \xrightarrow{\mathrm{rf}}$: $(d) \xrightarrow{\mathrm{rfe}} (a) \xrightarrow{\mathrm{s}_{\mathrm{lf}}} (b) \xrightarrow{\mathrm{rfe}} (c) \xrightarrow{\mathrm{s}_{\mathrm{lf}}} (d)$. If we have the guarantee that $\xrightarrow{\mathrm{s}_{\mathrm{lf}}}$ is A-cumulative *w.r.t.* $\xrightarrow{\mathrm{grf}_{2\setminus 1}}$ $(i.e.\ \forall xyz, (x \xrightarrow{\mathrm{grf}_{2\setminus 1}} y \wedge y \xrightarrow{\mathrm{s}_{\mathrm{lf}}} z) \Rightarrow x \xrightarrow{\mathrm{ab}_1} z)$, and if $\xrightarrow{\mathrm{s}_{\mathrm{lf}}}$ is included in $\xrightarrow{\mathrm{ghb}_1}$, we create a cycle in $\xrightarrow{\mathrm{ghb}_1}$: $(d) \xrightarrow{\mathrm{s}_{\mathrm{lf}}} (b) \xrightarrow{\mathrm{s}_{\mathrm{lf}}} (d)$.

Indeed, we show that if $\xrightarrow{\mathrm{s}_{\mathrm{lf}}}$ is A-cumulative *w.r.t.* $\xrightarrow{\mathrm{grf}_{2\setminus 1}}$ and compatible with $\xrightarrow{\mathrm{grf}_2}$, $\xrightarrow{\mathrm{ppo}_2}$ and $\xrightarrow{\mathrm{ghb}_1}$, then $\xrightarrow{\mathrm{s}_{\mathrm{lf}}}$ is covering:

Lemma 11 (A $\xrightarrow{\mathrm{s}_{\mathrm{lf}}}$ compatible with $\xrightarrow{\mathrm{ghb}_1}$ is covering)

$$\mathsf{acyclic}(\xrightarrow{\mathrm{s}_{\mathrm{lf}}} \cup \xrightarrow{\mathrm{grf}_2}) \wedge \mathsf{acyclic}(\xrightarrow{\mathrm{s}_{\mathrm{lf}}} \cup \xrightarrow{\mathrm{ppo}_2}) \wedge$$
$$\mathsf{acyclic}(\xrightarrow{\mathrm{s}_{\mathrm{lf}}} \cup \xrightarrow{\mathrm{ghb}_1}) \wedge \mathsf{AC}(\xrightarrow{\mathrm{s}_{\mathrm{lf}}}, \xrightarrow{\mathrm{grf}_{2\setminus 1}}) \Rightarrow \mathsf{covering}(\xrightarrow{\mathrm{c}_{\mathrm{lf}}}, \xrightarrow{\mathrm{s}_{\mathrm{lf}}})$$

Proof *Consider an execution* (E, X) *valid on* A_1 *and covered. Suppose by contradiction that there is a cycle in* $\xrightarrow{\mathrm{ghb}_2}$*, which is by definition a cycle in* $\xrightarrow{\mathrm{ws}} \cup \xrightarrow{\mathrm{fr}} \cup \xrightarrow{\mathrm{grf}_2} \cup \xrightarrow{\mathrm{ppo}_2}$*. Since* $\xrightarrow{\mathrm{ws}}$*,* $\xrightarrow{\mathrm{fr}}$ *and* $\xrightarrow{\mathrm{ppo}_1}$ *are included in* $\xrightarrow{\mathrm{ghb}_1}$*, this cycle is a cycle in* $\xrightarrow{\mathrm{ghb}_1} \cup \xrightarrow{\mathrm{ppo}_2 \setminus 1} \cup (\xrightarrow{\mathrm{grf}_2 \setminus 1}; \xrightarrow{\mathrm{ppo}_2})$*. Since* $\xrightarrow{\mathrm{slf}}$ *orders all fragile pairs, and is compatible with* $\xrightarrow{\mathrm{ppo}_2}$*, we know that* $\xrightarrow{\mathrm{ppo}_2 \setminus 1}$ *is included in* $\xrightarrow{\mathrm{slf}}$ *and* $(\xrightarrow{\mathrm{grf}_2 \setminus 1}; \xrightarrow{\mathrm{ppo}_2}) \subseteq (\xrightarrow{\mathrm{grf}_2 \setminus 1}; \xrightarrow{\mathrm{slf}})$*. Since* $\xrightarrow{\mathrm{slf}}$ *is A-cumulative,* $(\xrightarrow{\mathrm{grf}_2}; \xrightarrow{\mathrm{slf}})$ *is in* $\xrightarrow{\mathrm{ab}_1}$*, hence in* $\xrightarrow{\mathrm{ghb}_1}$*. Thus there is a cycle in* $\xrightarrow{\mathrm{slf}} \cup \xrightarrow{\mathrm{ghb}_1}$*, which contradicts their compatibility.* □

This means that if we place such barriers between the fragile pairs of a weak execution (*e.g.* (a) and (b) on P_0 and (c) and (d) on P_1), then this execution is either valid on A_2, or forbidden on A_1. We prove this result under the name *Barrier guarantee* in [AMSS].

10.3.3 Application to the Semantics of Barriers

This means that if there is an A-cumulative barrier after each fragile read and a non-cumulative barrier between each remaining fragile pair of an execution X valid on A_1, then X is valid on A_2. For example in the execution depicted in Fig. 10.1, we need an A-cumulative barrier between (a) and (b) on P_0 and between (c) and (d) on P_1.

10.3.3.1 Barriers Placement

We define the semantics and placement in the code that barriers should have to restore a stronger model from a weaker one. We define the predicate fb (*fully barriered*) on $A_1 \leq A_2$:

Definition 58 (Fully Barriered Execution)

$$A_1 \,.\, \mathsf{fb}_{A_2}(X) \;\triangleq\; \left((\xrightarrow{\mathrm{ppo}_2 \setminus 1}) \cup (\xrightarrow{\mathrm{grf}_2 \setminus 1}; \xrightarrow{\mathrm{ppo}_2}) \right) \subseteq \xrightarrow{\mathrm{ab}_1}$$

The fb predicate provides an insight to the strength that the barriers of the architecture A_1 should have to restore the stronger A_2. They should:

1. restore the pairs that are preserved in the program order on A_2 and not on A_1, which is a static property;

2. compensate for the fact that some writes may not be globally performed at once on A_1 while they are on A_2, which we model by (some subrelation of) $\xrightarrow{\mathrm{rf}}$ not being global on A_1 while it is on A_2; this is a dynamic property.

We prove indeed that the above condition on $\xrightarrow{\mathrm{ab}_1}$ restores A_2^{ϵ} from A_1:

Theorem 9 (Barriers Placement)

$$\forall A_1 A_2, (A_1 \leq A_2) \Rightarrow (\forall X, A_1 \,.\, \mathsf{valid}(X) \wedge A_1 \,.\, \mathsf{fb}_{A_2}(X) \Rightarrow A_2^{\epsilon} \,.\, \mathsf{valid}(X))$$

iriw

P_0	P_1	P_2	P_3
(a) `r1` $\leftarrow$ `x` `fence` (b) `r2` $\leftarrow$ `y`	(c) `r2` $\leftarrow$ `y` `fence` (d) `r1` $\leftarrow$ `x`	(e) `x` $\leftarrow$ `1`	(f) `y` $\leftarrow$ `2`

Observed? `0:r1=1; 0:r2=0; 1:r2=2; 1:r1=0;`

Figure 10.3: The **iriw** Example

Proof *Let X be an execution valid on A_1. Suppose $A_1 . \mathsf{fb}_{A_2}(X)$; let us show that X is valid on A_2. By definition, $\xrightarrow{\mathrm{ab}_1}$ is compatible with $\xrightarrow{\mathrm{ghb}_1}$ since included in it. Moreover, under the fully barriered hypothesis, $\xrightarrow{\mathrm{ab}_1}$ is A-cumulative. Hence, by Lem. 11, $\xrightarrow{\mathrm{ab}_1}$ is covering. Moreover, the fully barriered hypothesis ensures that X is covered by $\xrightarrow{\mathrm{ab}_1}$. Hence, by the definition of covering, if X is valid on A_1 then it is valid on A_2.* □

The static property of barriers is expressed by the condition $\xrightarrow{\mathrm{ppo}_{2\setminus 1}} \subseteq \xrightarrow{\mathrm{ab}_1}$. A barrier provided by A_1 should ensure that the events generated by a same processor are globally performed in program order if they are on A_2. In this case, it is enough to insert a barrier between the instructions that generate these events.

The dynamic property of barriers is expressed by the more involved condition $(\xrightarrow{\mathrm{grf}_{2\setminus 1}};\xrightarrow{\mathrm{ppo}_2}) \subseteq \xrightarrow{\mathrm{ab}_1}$. A barrier provided by A_1 should ensure store atomicity to the write events that have this property on A_2.

Thm. 9 states that, to restore A_2 from A_1, it suffices to insert an A-cumulative barrier between each pair of instructions such that the first one in the program order reads from a write which is to be globally performed on A_2 but is not on A_1.

Yet, it is enough to have $w \xrightarrow{\mathrm{ab}_1} r$ whenever $w \xrightarrow{\mathrm{grf}_{2\setminus 1}} r$ holds to restore store atomicity, *i.e.* a barrier ensuring that $\xrightarrow{\mathrm{rf}}$ is global. But then a processor holding such a barrier placed after r would wait until w is globally performed, then read again to ensure that r is globally performed after w. Our requirement is less costly: when $w \xrightarrow{\mathrm{rf}} r \xrightarrow{\mathrm{po}} m$, where r may not be globally performed after w is, inserting a barrier instruction between the instructions generating r and m only forces the processor generating r and m to delay m until w is globally performed.

Restoring SC Thm. 9 shows that inserting an A-cumulative barrier between all $\xrightarrow{\mathrm{po}}$ pairs restores SC:

Corollary 7 (Barriers Restoring SC)

$$\forall A\ X, (A . \mathsf{valid}(X) \wedge \mathsf{AC}(\xrightarrow{\mathrm{po}}, \xrightarrow{\mathrm{rf}})) \Rightarrow SC . \mathsf{valid}(X)$$

Consider *e.g.* the **iriw** test depicted in Fig. 10.3. The specified outcome may be the result of a non-SC execution on a weak architecture in the absence of barriers. Our A-cumulative barrier forbids this outcome, as shown in Fig. 10.4: if placed between each pair of reads on P_0 and P_1, not only does it prevent their reordering, but also ensures that the write (e) on P_2 (resp. (y) P_3) is globally performed before the second read (b) on P_0 (resp. (d) on P_1).

10.3.3.2 Weak Barrier Placement

We refine the study for two architectures that share the same store relaxation policy (*e.g.* Sparc TSO and PSO).

When two architectures A_1 and A_2 have the same policy *w.r.t.* the store atomicity and store buffer relaxations, which we model by $\xrightarrow{\text{grf}_1}=\xrightarrow{\text{grf}_2}$, there is no need for a barrier as powerful as above to restore A_2 from A_1: a barrier that only orders the events that surround it statically—that is, a non cumulative barrier, which action we model by $\mathsf{non\text{-}cumul}(X, \xrightarrow{\text{fenced}}) \triangleq \xrightarrow{\text{fenced}}$—is enough. Consider the wfb predicate, which states that the barriers provided by A_1 maintain the pairs that are preserved in the program order on A_2 but not on A_1:

Definition 59 (Weakly Fully Barriered Execution)

$$A_1 \,.\, \mathsf{wfb}_{A_2}(X) \;\triangleq\; \xrightarrow{\text{ppo}_{2\setminus 1}} \subseteq \xrightarrow{\text{ab}_1}$$

The same guarantee applies if A_2 hinders the store buffer relaxation by its preserved program order, *i.e.* when $\xrightarrow{\text{rfi}} \subseteq \xrightarrow{\text{ppo}_2}$—which is particular to SC:

Theorem 10 (Non-Cumulative Barriers Placement)

$$\forall A_1 A_2, ((A_1 \leq A_2) \wedge (\xrightarrow{\text{grf}_1} = \xrightarrow{\text{grf}_2} \vee (\xrightarrow{\text{rfi}} \subseteq \xrightarrow{\text{ppo}_2}))) \Rightarrow$$

$$(\forall X, A_1 \,.\, \mathsf{valid}(X) \wedge A_1 \,.\, \mathsf{wfb}_{A_2}(X) \Rightarrow A_2^{\epsilon} \,.\, \mathsf{valid}(X))$$

From TSO **to** SC As $\xrightarrow{\text{rfe}}$ are global in both TSO and SC, and SC hinders the store buffering relaxation by its $\xrightarrow{\text{ppo}}$ definition, it suffices by Thm. 10 to fence all pairs in $\xrightarrow{\text{ppo-sc}} \setminus \xrightarrow{\text{ppo-tso}} = \mathrm{WR}$ (where $\mathrm{WR} \;\triangleq\; (\mathbb{W} \times \mathbb{R}) \cap \xrightarrow{\text{po}}$) to restore SC from TSO:

Corollary 8 (Barriers Restoring SC **From** TSO**)**

$$\forall X, (\mathit{TSO} \,.\, \mathsf{valid}(X) \wedge \mathsf{NC}(\mathrm{WR})) \Rightarrow \mathit{SC} \,.\, \mathsf{valid}(X)$$

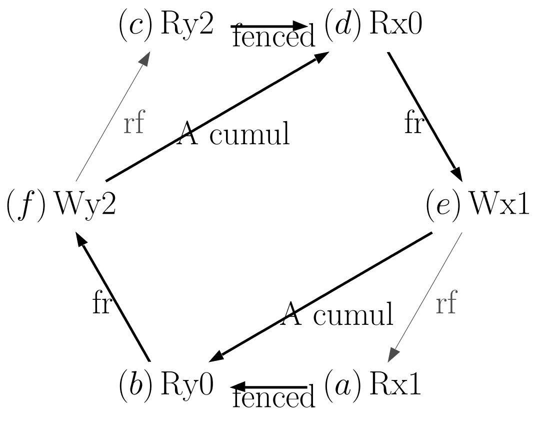


Figure 10.4: Study Of **iriw** With A-Cumulative Barriers

From *PSO* **to** TSO^{ϵ} We comment here on the two definitions of *PSO* given in Sparc documentations [spa94a]. *TSO* and *PSO* agree on both the store atomicity and the store buffering relaxations (see Sec. 4.2.2.2 and 4.2.2.3), which allows us to apply Thm. 10: TSO^{ϵ} is restored from *PSO* by inserting non-cumulative barriers between all $\xrightarrow{\text{ppo-tso}} \setminus \xrightarrow{\text{ppo-pso}} =$ WW pairs. Indeed, *TSO* is obtained from *PSO* by adding store-store barriers after each write [spa94a, V9, D.6, p. 296]:

> *The specification of Total Store Order (TSO) is that of Partial Store Order (PSO) with the additional requirement that all memory transactions with store semantics are followed by an implied MEMBAR [i.e. memory barrier] #StoreStore.*

10.3.4 Lock-free guarantee

We now refine this result: we show that we only have to check the barriers' placement on all the executions valid on A_2 (instead of A_1 in Thm. 9).

We define the lock-free guarantee to stipulate that if all the fragile pairs of the executions valid on A_2 are arbitrated by $\xrightarrow{\text{s}_{\text{lf}}}$, then there is no other execution:

Definition 60 (Lock-Free Guarantee)

$$\begin{array}{l} A_1 \,.\, \mathrm{lfg}(A_2) \triangleq \\ \quad (\forall EX, A_2 \,.\, \mathsf{valid}(E,X) \Rightarrow \mathsf{covered}_{\mathrm{LF}}(E,X)) \Rightarrow \\ \qquad\qquad (\forall EX, \mathsf{stable}_{\mathsf{A_1},\mathsf{A_2},(}\, E,X)) \end{array}$$

We have shown in Lem. 11 that a $\xrightarrow{\text{s}_{\text{lf}}}$ which is A-cumulative and compatible with $\xrightarrow{\text{grf}_2}$, $\xrightarrow{\text{ppo}_2}$ and $\xrightarrow{\text{ghb}_1}$, is covering. Hence Thm. 8 ensures that if $\xrightarrow{\text{s}_{\text{lf}}}$

Name	Code	Comment
plain load	`lwz r1,0,r2`	loads into `r1` from the address in `r2`
plain store	`stw r1,0,r2`	stores from `r1` into the address in `r2`
load reserve	`lwarx r1,0,r2`	loads from the address in `r2` into `r1` and reserves the address in `r2`
store conditional	`stwcx.  r1,0,r2`	checks if the address in `r2` is reserved; if so, stores from `r1` into this address; if not, fails
branch not equal	`bne L`	branches to `L` if condition code register (`CR0`) encodes disequality
compare	`cmpw r4, r6`	compares values in `r4` and `r6` and stores result into `CR0`
isync	`isync`	when placed after a `bne`, forms a RW, RR non-cumulative barrier
lwsync	`lwsync`	RW, RR, WW A- and B-cumulative barrier
sync	`sync`	RW, RR, WW, WR A- and B-cumulative barrier

Figure 10.5: Table of the Power assembly instructions used in this chapter

satisfies these properties and is well-founded, our lock-free guarantee holds:

Corollary 9 (Compatibility of $\xrightarrow{\mathrm{s_{lf}}}$ with A_1 ensures stability)

$$\mathsf{wf}(\xrightarrow{\mathrm{c_{lf}}}, \xrightarrow{\mathrm{s_{lf}}}) \wedge \mathsf{acyclic}(\xrightarrow{\mathrm{s_{lf}}} \cup \xrightarrow{\mathrm{grf_2}}) \wedge \mathsf{acyclic}(\xrightarrow{\mathrm{s_{lf}}} \cup \xrightarrow{\mathrm{ppo_2}}) \wedge$$
$$\mathsf{acyclic}(\xrightarrow{\mathrm{s_{lf}}} \cup \xrightarrow{\mathrm{ghb_1}}) \wedge \mathsf{AC}(\xrightarrow{\mathrm{grf_{2\setminus 1}}}, \xrightarrow{\mathrm{s_{lf}}}) \Rightarrow A_1 \,.\, \mathrm{lfg}(A_2)$$

This states that it is enough to place barriers in an execution (E, X) valid on A_2, between each pair of $\xrightarrow{\mathrm{ppo_{2\setminus 1}}}$ and after each read reading from a write that should be atomic on A_2 but is not on A_1. For example in Fig. 10.2(b), we have to place an A-cumulative barrier between (c) and (d) on P_1 and a non-cumulative one between (a) and (b) on P_0. In Fig. 10.2(c), we have to place two non-cumulative barriers, one between (a) and (b) on P_0 and another one between (c) and (d) on P_1. This ensures that there can be no other execution but the SC ones depicted in Fig. 10.2. Once again, the pairs to be arbitrated are the same as in Fig. 10.1(b), but we only had to examine the SC executions to find them.

10.4 Synchronisation idioms

We examine here the semantics of some synchronisations primitives. We study the locks and read-modify-writes as described in Power's documentation [pow09]. We show that these constructs ensure a SC behaviour to Power programs, as they induce covering relations. We give a table of the instructions in Fig. 10.5.

```
       loop:
(a1)   lwarx r1,0,r5
       [...]
(a2)   stwcx.  r2,0,r5
 (b)   bne loop
```

Figure 10.6: A generic read-modify-write in Power assembly

10.4.1 Atomicity

Fig. 10.6 gives a generic Power read-modify-write. In between the `lwarx` (a_1) and the `stwcx.` (a_2), the code is left to the choice of the programmer, supposing there are no other `lwarx` and `stwcx.` between (a_1) and (a_2). The `lwarx` (a_1) loads from its source address (in register `r5`) and *reserves* it. Any subsequent store to the reserved address from another processor and any subsequent `lwarx` from the same processor invalidates the reservation. The `stwcx.` (a_2) checks if the reservation is still valid; if so, it is said to be *successful*: it stores into the reserved address and the code exits the loop. Otherwise, the `stwcx.` does not perform any store and the code loops. Thus these instructions ensure *atomicity* to the operations they surround, as no other processor can write to the reserved location between the `lwarx` and the successful `stwcx.`.

We distinguish the read and write events issued by such instructions from the plain ones: we write $\mathbb{R}^*$ (resp. $\mathbb{W}^*$) for the subset of $\mathbb{R}$ (resp. $\mathbb{W}$) issued by a `lwarx` (resp. a successful `stwcx.`), and define an atomic pair as follows:

Definition 61 (Atomic pair)

$$\begin{aligned}\mathsf{atom}(r, w, \ell) \triangleq\ & r \in \mathbb{R}^* \wedge w \in \mathbb{W}^* \wedge \mathsf{loc}(r) = \mathsf{loc}(w) = \ell \wedge \\ & r = \max_{\xrightarrow{\text{po}}}(\{m \mid m \in (\mathbb{R}^* \cup \mathbb{W}^*) \wedge m \xrightarrow{\text{po}} w\}) \wedge \\ & \neg(\exists w', \mathsf{proc}(w') \neq \mathsf{proc}(r) \wedge \mathsf{loc}(w') = \ell \wedge r \xrightarrow{\text{fr}} w' \xrightarrow{\text{ws}} w)\end{aligned}$$

Thus two events r and w form an atomic pair $w.r.t.$ a location ℓ if:

- w was issued by a successful `stwcx.` to ℓ, and
- r was issued by the last `lwarx` (in $\xrightarrow{\text{po}}$) from ℓ before the `stwcx.` that issued w, and
- no other processor wrote to ℓ between r and w.

10.4.2 Locks

Atomic pairs are used *e.g.* in *lock* and *unlock* primitives [pow09, App. B]. The idiomatic Power implementation of lock is shown in Fig. 10.7(a). The

Initially `r3 =` ℓ, `r4 = 0` and `r5 = 1`

```
       loop:
(a1)   lwarx r6,0,r3
(b)    cmpw r4,r6
(c)    bne loop
(a2)   stwcx.  r5,0,r3
(d)    bne loop
(e)    isync
       [...]
```

(a) Lock

```
       [...]
(f)    lwsync
(g)    stw r4,0,r3
```

(b) Unlock

Figure 10.7: Lock and Unlock in Power

lines (a_1) to (a_2) match the scheme of Fig. 10.6; this sequence loops until it acquires the lock. The acquisition is followed by a sequence `bne;isync` (lines (d) and (e)), forming a barrier, which ensures that no access inside the critical section can be speculated before the lock primitive.

Fig. 10.7(b) shows the implementation of unlock. It starts with a `lwsync` barrier (line (f)), ensuring that no access inside the critical section can be delayed after the unlock. The barrier is followed by a store to the lock variable ℓ (line (g)), which frees the lock.

Semantics We formalise these notions as follows. A read r takes a lock ℓ if it reads from ℓ and forms an atomic pair with a write w to ℓ. The next write event to ℓ (if any) in program order after a lock acquisition frees the lock. Such a write writes to the lock variable, and is after r in the program order:

Definition 62 (Taken and Free)

$$\begin{aligned}&\mathsf{taken}(\ell, r) \triangleq \exists w, \mathsf{atom}(r, w, \ell)\\&\mathsf{free}(\ell, r, w) \triangleq r \xrightarrow{\text{po}} w \wedge \mathsf{taken}(\ell, r) \wedge \mathsf{loc}(w) = \ell\end{aligned}$$

A lock acquisition consists of a taken operation (see Fig. 10.7, lines (a_1) to (a_2)) followed by an *import barrier* [pow09, p. 721] (lines (d) and (e)), whose properties we study in the next paragraph. An unlock consists of an *export barrier* [pow09, p. 722] (line (f)), also studied next, followed by a write freeing the lock (line (g)):

Definition 63 (Lock and Unlock)

$$\mathsf{Lock}(\ell, r) \triangleq \mathsf{taken}(\ell, r) \wedge \mathsf{import}(r)$$

$$\mathsf{Unlock}(\ell, r, w) \triangleq \mathsf{free}(\ell, r, w) \wedge \mathsf{export}(w)$$

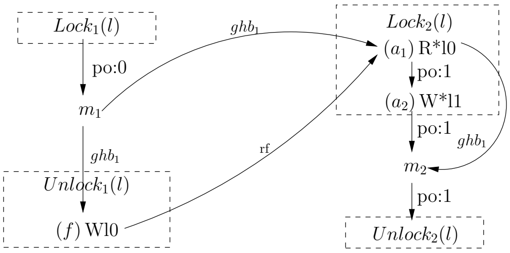


Figure 10.8: Opening Lock and Unlock

A *critical section* consists of a lock and an unlock with the same variable ℓ, and the events in $\xrightarrow{\text{po}}$ between the lock's barrier and the unlock's one:

Definition 64 (Critical Section)

$$\begin{aligned}\text{cs}(\mathcal{E}, \ell, r, w) \triangleq\ & \mathsf{Lock}(\ell, r)\ \wedge \\ & \mathcal{E} = \{e \mid r \xrightarrow{\text{po}} e \xrightarrow{\text{po}} w\} \wedge \mathsf{Unlock}(\ell, r, w)\end{aligned}$$

We write $\mathsf{loc}(\text{cs})$ for the location of a given critical section cs. Two critical sections cs_1 and cs_2 with the same location ℓ are *serialised* if cs_2 reads from cs_1, as depicted in Fig. 10.8: the left-hand side of the picture is the first critical section cs_1, composed of a lock acquisition $\mathsf{Lock}_1(\ell)$, an event m_1 and an unlock $\mathsf{Unlock}_1(\ell)$, which writes into ℓ *via* the write (f). The second critical section cs_2 is on the right-hand side: the read (a_1) of its lock acquisition $\mathsf{Lock}_2(\ell)$ reads from (f). Thus, cs_1 and cs_2 are serialised if cs_2 Lock's read (written $\text{R}(\text{cs}_2)$) reads from cs_1 Unlock's write (written $\text{W}(\text{cs}_1)$):

Definition 65 (Serialisation of critical sections)

$$\text{cs}_1 \xrightarrow{\text{css}_\ell} \text{cs}_2 \triangleq \mathsf{loc}(\text{cs}_1) = \mathsf{loc}(\text{cs}_2) = \ell \wedge \text{W}(\text{cs}_1) \xrightarrow{\text{rf}} \text{R}(\text{cs}_2)$$

Given a location ℓ, two events m_1 and m_2 are in $\xrightarrow{\text{lock}_\ell}$ if they are in two serialised critical sections (as in Fig. 10.8), or m_1 is in $\xrightarrow{\text{lock}_\ell}$ with an event itself in $\xrightarrow{\text{lock}_\ell}$ with m_2 ($m \in \text{cs}$ ensures m is between cs import and export barriers in $\xrightarrow{\text{po}}$):

Definition 66 (Per location lock relation)

$$\begin{aligned}& m_1 \xrightarrow{\text{lock}_\ell} m_2 \triangleq \\ & \quad (\exists\, \text{cs}_1\, \text{cs}_2, m_1 \in \text{cs}_1 \wedge m_2 \in \text{cs}_2 \wedge \text{cs}_1 \xrightarrow{\text{css}_\ell} \text{cs}_2) \vee \\ & \quad (\exists m, m_1 \xrightarrow{\text{lock}_\ell} m \xrightarrow{\text{lock}_\ell} m_2)\end{aligned}$$

Finally, two events m_1 and m_2 are in $\xrightarrow{\text{lock}}$ if: $\exists \ell, m_1 \xrightarrow{\text{lock}_\ell} m_2$.

iriw

P_0	P_1	P_2	P_3
(a) `r1` ← `x` (b) `r2` ← `y`	(c) `r2` ← `y` (d) `r1` ← `x`	(e) `x` ← `1`	(f) `y` ← `2`

Observed? `0:r1=1`; `0:r2=0`; `1:r2=2`; `1:r1=0`;

Figure 10.9: The **iriw** example

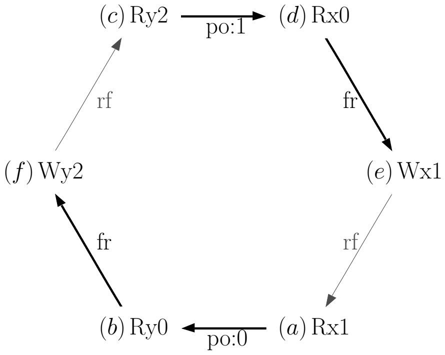


Figure 10.10: A non-SC execution of **iriw**

Barriers We define an import barrier to prevent any event to float above a read issued by a `lwarx`: in Fig. 10.8, the event m_2 in cs_2 is in $\xrightarrow{\text{ghb}_1}$ with the read (a_1) from its Lock's `lwarx`. We define an export barrier to force all the events inside a critical section to be globally performed before the next lock primitive takes the lock: in Fig. 10.8, the event m_1 in cs_1 is in $\xrightarrow{\text{ghb}_1}$ with the read (a_1) of cs_2's Lock. This means that we define an export barrier to be B-cumulative, but only *w.r.t.* read-from maps whose target is a read issued by a `lwarx`. Formally, we have:

Definition 67 (Import and export barriers)

$$\begin{aligned} &\mathsf{import}(r) \triangleq \forall m, (r \xrightarrow{\text{po}} m) \Rightarrow (r \xrightarrow{\text{ab}_1} m) \\ &\mathsf{export}(w) \triangleq \forall rm, (r \in \mathbb{R}^* \wedge (m \xrightarrow{\text{po}} w \xrightarrow{\text{rf}} r)) \Rightarrow (m \xrightarrow{\text{ab}_1} r) \end{aligned}$$

DRF guarantee If the import and export barriers of our critical sections have the above semantics, there is no nested critical sections and no event of a critical section accesses its lock variable, then we show that $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$ is acyclic:

Lemma 12 (Compatibility of $\xrightarrow{\text{lock}}$ with $\xrightarrow{\text{po}}$)

$$\forall EX, A_1 \,.\, \mathsf{valid}(E, X) \Rightarrow \mathsf{acyclic}(\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}})$$

Proof *Suppose by contradiction a cycle in* $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$. *This cycle is a cycle in* $((\xrightarrow{\text{lock}})^+; \xrightarrow{\text{po}})$ *since* $\xrightarrow{\text{po}}$ *is transitive. Let us show by induction that any path of* $(\xrightarrow{\text{lock}})^+; \xrightarrow{\text{po}}$ *from an event x to an event y is a path in* $(\xrightarrow{\text{ghb}_1})^+$. *Hence, a cycle is in* $(\xrightarrow{\text{lock}})^+; \xrightarrow{\text{po}}$ *is a cycle in* $(\xrightarrow{\text{ghb}_1})^+$, *which contradicts the validity of (E, X) on A_1.*

Consider the base case with three events $m_1 \xrightarrow{\text{lock}} m_2 \xrightarrow{\text{po}} m_3$. *Let us do an induction over* $m_1 \xrightarrow{\text{lock}} m_2$. *Consider the base case where m_1 and m_2 belong respectively to the critical sections* cs_1 *and* cs_2, *such that* $\text{cs}_1 \xrightarrow{\text{css}} \text{cs}_2$.

In this case, m_3 is in $\xrightarrow{\text{po}}$ *after* cs_2*'s import barrier, which prevents any event to float above the read issued by* cs_2*'s* `lwarx`. *Thus we have* $\text{R}(\text{cs}_2) \xrightarrow{\text{ab}_1} m_3$, *hence* $\text{R}(\text{cs}_2) \xrightarrow{\text{ghb}_1} m_3$. *Moreover, m_1 is in* $\xrightarrow{\text{po}}$ *before* cs_1*'s export barrier (i.e.* $m_1 \xrightarrow{\text{ab}_1} \text{W}(\text{cs}_1)$*) hence* $m_1 \xrightarrow{\text{ghb}_1} \text{W}(\text{cs}_1)$. *Since* $\text{W}(\text{cs}_1) \xrightarrow{\text{rfe}} \text{R}(\text{cs}_2)$, *by B-cumulativity of* cs_1*'s export barrier, we have* $m_1 \xrightarrow{\text{ab}_1} \text{R}(\text{cs}_2)$, *hence* $m_1 \xrightarrow{\text{ghb}_1} \text{R}(\text{cs}_2)$. *Thus* $m_1 (\xrightarrow{\text{ghb}_1})^+ m_3$.

The transitive cases follows by induction. □

This result means, following Lem. 10, that $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$ is a covering synchronisation relation for the competing accesses. We show that $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$ forms a well founded relation:

Lemma 13 ($\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$ is well-founded)

$$\mathsf{wf}(\overset{\text{c}_{\text{drf}}}{\leftrightarrow}, \xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}})$$

Proof *Let (E, X) be an execution valid on A_1 and uncovered: there is a competing pair* $x \overset{\text{c}_{\text{drf}}}{\leftrightarrow} y$ *which is not arbitrated by* $\xrightarrow{\text{lock}} \cup \xrightarrow{\text{po}}$. *Hence one of them at least is not protected by any critical section: suppose it is x (we omit the symmetrical case). We know by definition that* $x \overset{\text{c}_{\text{drf}}}{\leftrightarrow} y \Rightarrow \neg(y \xrightarrow{\text{po}} x)$. *Hence, since* $A_2 \,.\, \mathsf{ppo}(E) \subseteq \mathsf{po}(E)$, *we have* $(x \overset{\text{c}_{\text{drf}}}{\leftrightarrow} y) \Rightarrow \neg((y, x) \in A_2 \,.\, \mathsf{ppo}(E))$. *Therefore,* $((x, y) \cup A_2 \,.\, \mathsf{ppo}(E))^+$ *is a partial order over* $\mathsf{evts}(E)$ *compatible with* $A_2 \,.\, \mathsf{ppo}(E)$. *We know by Lem.* **??** *that from any linear extension of this order, we can build an execution witness Y associated with E and valid on A_2. Moreover, (x, y) belongs to (E, Y), and since we kept exactly the events of E, there are no events forming a critical section to protect x in Y either.*

Hence by Cor. 6 the critical sections defined above provide the DRF guarantee as detailed in the following. We take the $\xrightarrow{\text{sync}}$ relation of the DRF_0 model defined at Sec. 10.2 to be as follows and compatible with $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{rf}}$:

$$\xrightarrow{\text{sync}} \triangleq \{(x, y) \mid \exists cs_1 \neq cs_2, \mathsf{loc}(\text{cs}_1) = \mathsf{loc}(\text{cs}_2) \wedge x \in \text{cs}_1 \wedge y \in \text{cs}_2\}$$

This means that two events are in $\xrightarrow{\text{sync}}$ when they belong to distinct critical sections with the same variable. $\xrightarrow{\text{sync}}$ is well-founded since the competing pairs and the $\xrightarrow{\text{sync}}$ definitions are independent of any execution.

We show that two events $x \xrightarrow{\text{sync}} y$ are ordered by $\xrightarrow{\text{lock}}$:

Lemma 14 ($\xrightarrow{\text{lock}}$ orders $\xrightarrow{\text{sync}}$)

$$\forall xy, x \xrightarrow{\text{sync}} y \Rightarrow x \xrightarrow{\text{lock}} y \vee y \xrightarrow{\text{lock}} x$$

Proof *Consider two events $m_1 \overset{\text{sync}}{\rightarrow} m_2$. We write* cs_1 *(resp.* cs_2*) for the critical section to which m_1 (resp. m_2) belongs. If* cs_2 *reads from* cs_1 *(resp.* cs_1 *from* cs_2*), we know $x \overset{\text{lock}}{\rightarrow} y$ (resp. $y \overset{\text{lock}}{\rightarrow} x$). Otherwise, there exists a write w_1 (resp. w_2) from which* cs_1 *(resp.* cs_2*) reads. The writes w_1 and w_2 are to the same location, since they are in $\overset{\text{rf}}{\rightarrow}$ with the reads from* cs_1 *and* cs_2*, which have the same location. Since $\overset{\text{ws}}{\rightarrow}$ is a total order on the writes to the same location, we have $w_1 \overset{\text{ws}}{\rightarrow} w_2$ or $w_2 \overset{\text{ws}}{\rightarrow} w_1$. Suppose $w_1 \overset{\text{ws}}{\rightarrow} w_2$.* cs_1 *reads from w_1, therefore* $\mathrm{W}(\mathrm{cs}_1)$ *(the write releasing the lock of* cs_1*) occurs in $\overset{\text{ws}}{\rightarrow}$ after w_1. Since* $\mathrm{W}(\mathrm{cs}_1)$ *and w_2 have the same location, we have* $\mathrm{W}(\mathrm{cs}_1) \overset{\text{ws}}{\rightarrow} w_2 \vee w_2 \overset{\text{ws}}{\rightarrow} \mathrm{W}(\mathrm{cs}_1)$*. The second case is impossible since no write to the lock variable can occur between the read and the write of a critical section. In the first case, we have $x \overset{\text{lock}}{\rightarrow} y$.* □

Since $\overset{\text{sync}}{\rightarrow}$ is compatible with $\overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow}$, it is also compatible with $\overset{\text{lock}}{\rightarrow}$. Indeed, $\overset{\text{lock}}{\rightarrow}$ is induced by a $\overset{\text{rf}}{\rightarrow}$ relation between the write of the first critical section and the read of the second. Therefore, $\overset{\text{sync}}{\rightarrow}$ is included in $\overset{\text{lock}}{\rightarrow}$:

Lemma 15 ($\overset{\text{sync}}{\rightarrow}$ is included in $\overset{\text{lock}}{\rightarrow}$)

$$\overset{\text{sync}}{\rightarrow} \subseteq \overset{\text{lock}}{\rightarrow}$$

Proof *Consider two events x and y in $\overset{\text{sync}}{\rightarrow}$. Since $\overset{\text{lock}}{\rightarrow}$ orders $\overset{\text{sync}}{\rightarrow}$, we know $x \overset{\text{lock}}{\rightarrow} y$ or $y \overset{\text{lock}}{\rightarrow} x$. Since $\overset{\text{sync}}{\rightarrow}$ and $\overset{\text{lock}}{\rightarrow}$ are compatible, we have $x \overset{\text{lock}}{\rightarrow} y$.* □

We know by Lem. 12 that $\overset{\text{lock}}{\rightarrow}$ and $\overset{\text{po}}{\rightarrow}$ are compatible. Since $\overset{\text{sync}}{\rightarrow}$ is included in $\overset{\text{po}}{\rightarrow}$, we know that $\overset{\text{sync}}{\rightarrow}$ and $\overset{\text{po}}{\rightarrow}$ are compatible. Since $\overset{\text{sync}}{\rightarrow}$ is well founded, we know by Cor. 6 that $(\overset{\text{sync}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow})^+$ provides the DRF guarantee.

Thus, given a program, we only have to place such critical sections on the competing accesses of the strong executions of this program to ensure that all its executions are strong.

Discussion Our import barrier allows events to be delayed so that they are performed inside the critical section. Our export barrier allows the events after the unlock to be speculated before the lock is released. Such relaxed semantics already exist for high-level lock and unlock primitives [Boea, Sev08].

Our study does not contradict the Power documentation [pow09, p. 721 and 722], but suggests that we do not need all the features of the import barrier appearing in the documentation, and supposes an undocumented feature of the export barrier, though it has not been invalidated by our experiments.

Indeed, the import barrier is a sequence `bne;isync` (*i.e.* a read-read, read-write non-cumulative barrier) or a `lwsync` (*i.e.* a read-read, read-write, write-write A- and B-cumulative barrier) [pow09, p.721]. We show that we do not need cumulativity, nor to maintain write-write pairs. We only need to maintain read-read and read-write pairs. Moreover, we can even restrict these pairs to pairs where the first read corresponds to a `lwarx`.

The export barrier is a `sync` (*i.e.* an A- and B-cumulative barrier maintaining all pairs) or a `lwsync` [pow09, p. 722]. We show that we only need a B-cumulative barrier towards reads issued by a `lwarx`, therefore a `sync` is unnecessarily costly. Moreover, `lwsync` is cumulative except from a write to a read, whereas we require the export barrier to be B-cumulative from any event to a read issued by a `lwarx`. Although a `lwsync` is not B-cumulative towards plain reads, its implementations appear experimentally to treat reads issued by a `lwarx` specially, and be B-cumulative towards such reads. Since this is undocumented, this raises the question of whether this implementation of locks actually ensures the DRF guarantee, and even if it does, for which reasons. The lack of details in the documentation leaves the answer unclear.

10.4.3 Lock-free synchronisation

Consider the **iriw** example [BA] given in Fig. 10.9; we also give an associated non-SC execution in Fig. 10.10. We know by Lem. 11 that we can restore SC by placing two A-cumulative barriers between the fragile pairs ($i.e.$ between (a) and (b) on P_0, and between (c) and (d) on P_1).

However, the cumulativity of a barrier may be challenging to implement or too costly in practice. We propose a mapping of certain reads to read-modify-write primitives (RMW) (such as the one given in Fig. 10.6), and show that this mapping restores a strong architecture from a weaker one without using cumulativity, and by checking the strong executions only.

Consider the execution of **iriw** given in Fig. 10.11. This execution is valid on SC since it does not exhibit any cycle in $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$. We give in Fig. 10.12(a) the same execution, where we have replaced the fragile reads (a) and (c) by RMW primitives: we say these fragile reads are *protected* (a notion defined below).

In fact, we have chosen a particular kind of RMW for this example: we use *fetch and no-op* (FNO) primitives. The FNO idiom of [pow09, p.719] is a RMW such as in Fig. 10.6, with no code between the `lwarx` and `stwcx.`, such that the target of the `lwarx` is the source of the `stwcx.`. Hence, a FNO implements an atomic read. Yet, our results hold for any kind of RMW.

We show indeed that when the fragile reads are protected, we do not need cumulative barriers anymore, but just non-cumulative ones. The intuition is that if a read is protected by a RMW, then the RMW compensates the need for cumulativity by enforcing enough order to the write from which the protected read reads.

Protecting the fragile reads with RMW We consider that two events r and w form a RMW $w.r.t.$ a location ℓ if they form an atomic pair $w.r.t.$ ℓ ($i.e.$ the code in Fig. 10.6 does not loop), or there is a read r' after r in the program order forming an atomic pair $w.r.t.$ ℓ with w, such that r' is the last read issued by the loop before the `stwcx.` succeeds ($i.e.$ the code in Fig. 10.6 loops). We do not consider the case where the loop never returns.

For example in Fig. 10.12(b), we open up the FNO box protecting the read (a) from x on P_0. We suppose here that the FNO is immediately successful, *i.e.* the code in Fig. 10.6 does not loop. Hence we expand the FNO event (a) on P_0 to the r^* (a_1) (from the `lwarx`) in program order with the w^* (a_2) (from the successful `stwcx.`). Formally, we define a RMW as follows:

Definition 68 (Read-modify-write)

$$\begin{aligned}\mathsf{rmw}(r, w, \ell) \triangleq\ & \mathsf{atom}(r, w, \ell) \vee \\ & (\exists r', r \xrightarrow{\text{po}} r' \wedge \mathsf{loc}(r) = \mathsf{loc}(r') \wedge \mathsf{atom}(r', w, \ell))\end{aligned}$$

We define a read to be *protected* when it is issued by the `lwarx` of a RMW immediately followed in program order by a non-cumulative barrier; an execution X is protected when its fragile reads are:

Definition 69 (Protected read and execution)

$$\begin{aligned}\mathsf{protected}(r) &\triangleq \exists w, \mathsf{rmw}(r, w, \mathsf{loc}(r)) \wedge \\ &\qquad (\forall m, w \xrightarrow{\text{po}} m \Rightarrow w \xrightarrow{\text{ab}_1} m) \\ \mathsf{protected}(E, X) &\triangleq \forall r, \mathsf{fragile}(r) \Rightarrow \mathsf{protected}(r)\end{aligned}$$

Consider Fig. 10.12(b): the read (a_1) and the write (a_2) are in the program order and to the same location, hence we have $(a_1) \xrightarrow{\text{fr}} (a_2)$. This means that the write (e) from which (a_1) reads hits the memory before (a_2) does, *i.e.* $(e) \xrightarrow{\text{ws}} (a_2)$. Hence the sequence $(e) \xrightarrow{\text{rf}} (a) \xrightarrow{\text{po}} (b)$ is covered by the sequence $(e) \xrightarrow{\text{ws}} (a_2) \xrightarrow{\text{po}} (b)$. Thus, we compensate the lack of store atomicity of (e) (*i.e.* the fact that $(e) \xrightarrow{\text{rfe}} (a)$ is not global) without using cumulativity, but by using the write serialisation between (e) and the write (a_2) from the RMW. Formally, we show that any sequence $w \xrightarrow{\text{grf}_2\setminus 1} r \xrightarrow{\text{ppo}_2} m$ where r is protected is in $\xrightarrow{\text{ws}};\xrightarrow{\text{ghb}_1}$, hence in $(\xrightarrow{\text{ghb}_1})^+$:

Lemma 16 (Protected reads property)

$$\forall wrm, (\mathsf{protected}(r) \wedge w \xrightarrow{\text{grf}_2\setminus 1} r \xrightarrow{\text{ppo}_2} m) \Rightarrow w \xrightarrow{\text{ws}};\xrightarrow{\text{ghb}_1} m$$

Proof *Since r is protected, there are r' and w' such that $\mathsf{rmw}(r', w', \mathsf{loc}(r))$ where r' is r or a subsequent read in $\xrightarrow{\text{po}}$. In both cases, we have $w \xrightarrow{\text{ws}} w'$. Moreover, since there is a barrier between w' and m (i.e. $w' \xrightarrow{\text{ab}_1} m$), we know that $w' \xrightarrow{\text{ghb}_1} m$.*

We saw in Sec. 10.3 that covering the fragile pairs enforces stability. If we protect the fragile reads, the only remaining fragile pairs are the ones in $\xrightarrow{\text{ppo}_2\setminus 1}$. For example in Fig. 10.12(a), we have $(e) \xrightarrow{\text{ws}} (a_2)$. Moreover, we have $(b) \xrightarrow{\text{fr}} (f)$ since (b) reads from the initial state, thus: $(e) \xrightarrow{\text{ws}} (a_2) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (f)$. Similarly on P_1 we have $(f) \xrightarrow{\text{ws}} (c_2) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (e)$, hence a cycle in $\xrightarrow{\text{ws}} \cup \xrightarrow{\text{fr}} \cup \xrightarrow{\text{po}}$. Since $\xrightarrow{\text{ws}}$ and $\xrightarrow{\text{fr}}$ are global, we know that to invalidate this

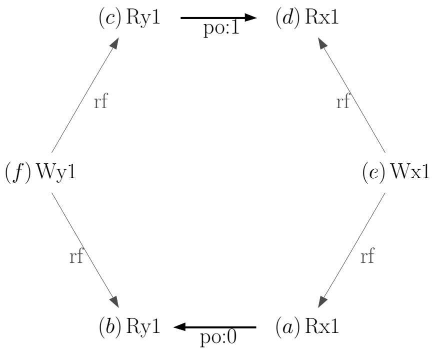
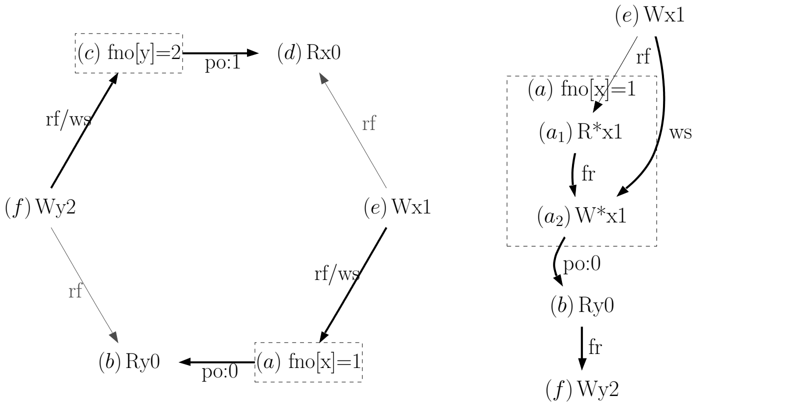


Figure 10.11: A SC execution of **iriw**



(a) Synthetic execution witness (b) Opening FNO on P_0

Figure 10.12: Study of **iriw** after FNO transformation

cycle, we only need to order globally (*e.g.* by a barrier) the accesses (a_2) and (b) on P_0 and (c_2) and (d) on P_1.

Indeed, if a strong execution is protected, non-cumulative barriers placed between the remaining fragile pairs in $\xrightarrow{\text{ppo}_{2\setminus 1}}$ ensure stability:

Lemma 17 (A protected execution is stable)

$$\forall EX, A_2 \,.\, \mathsf{valid}(E,X) \wedge \mathsf{protected}(E,X) \wedge \\ (\xrightarrow{\text{ppo}_{2\setminus 1}} \subseteq \xrightarrow{\text{ab}_1}) \Rightarrow \mathsf{stable}_{\mathsf{A}_1,\mathsf{A}_2,(}E,X)$$

Proof *Barriers are by definition compatible with* $\xrightarrow{\text{ghb}_1}$*. The A-cumulativity is handled by the protection of the fragile reads as shown in Lem. 16. Finally, the barriers order globally the remaining fragile pairs. Hence the barriers induce a covering synchronisation relation by Lem. 11. Since it is trivially well-founded, Cor. 9 applies.* □

Discussion A mapping of the stores into RMW was proposed for x86 [BA] (where there are no fragile reads) to provide a SC semantics to C++ atomics. For models where reads may be fragile (*e.g.* Power) the existence of "more efficient mappings (than the use of locks)" is unclear, as exposed in [AB]. Our study suggests that our mapping could be more efficient than locks, since it removes the need for cumulative barriers, which are necessary for locks. However, mapping reads to RMW introduces additional stores (issued by `stwcx.`), which may impair the performance. The trade-off between these two mappings is unclear.

Moreover, since there is no full non-cumulative barrier in Power, we have to use cumulative barriers even though we do not need all their features. *Dependencies* (*e.g. via* register index) between read-read and read-write pairs can be used to simulate non-cumulative barriers, hence spare the cost of cumulativity on such pairs if the memory model ensures that such dependencies are global, as does *e.g.* Power [AMSS]. The existence of such a solution for write-read and write-write pairs is unclear.

All reads and writes could be mapped into RMW (using FNO for reads and fetch-and-store [pow09, p. 719] for writes, so as to preserve the semantics). The documentation stipulates indeed that "a processor has at most one reservation at any time" [pow09, p. 663]. Hence two RMW on the same processor in program order may be preserved in this order, because the writes issued by their `stwcx.`, though to different locations, would be ordered by a dependency over the reservation. In cases of hyperthreading for example, it could mean that several threads running on the same processor share a global dependency *via* the reservation of this processor. This has not been invalidated by our experiments, but the documentation does not state whether this dependency is global, or even exists.

Chapter 11

Stability

The data race free and lock-free guarantees may require more synchronisation than necessary. Consider *e.g.* the **iriw** example given in Fig. 7.1. If we add a write (g) to a fresh variable z after (in program order) the write (e) to x on P_2, (e) and (g) may not be preserved in program order. Yet, there is no need to maintain them, since they do not contribute to the cycle we want to forbid.

Hence, in this section, we refine our study in the style of D. Shasha and M. Snir [SS], in order to minimise the number of synchronisation mechanisms used to enforce the behaviour of the stronger architecture. To do so, we define the *minimal cycles* of an execution, which are the cycles we want to prevent.

We then show that it is enough to synchronise the minimal cycles (instead of all cycles previously) of an execution to ensure its stability. Finally, we show that, given an event structure, all the associated executions are stable if and only if the event structure does not contain any minimal cycle.

11.1 Minimal cycles

We consider that a relation $\xrightarrow{\sigma}$ is a *cycle* (written $\mathsf{cycle}(\xrightarrow{\sigma})$) when $\xrightarrow{\sigma}$ is non-empty, symmetric and its transitive closure is total. We define a *minimal cycle* as a cycle $\xrightarrow{\sigma}$ such that there is no other cycle $\xrightarrow{\sigma}'$ distinct from $\xrightarrow{\sigma}$ and included in $\xrightarrow{\sigma}$.

11.1.1 Violations

Given an event structure E, we consider a cycle to be a *violation* of A_2 *w.r.t.* A_1 when it is a cycle in $(\overset{\mathrm{c_{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{ppo_2}})^+$ (*i.e.* $\xrightarrow{\sigma} \subseteq (\overset{\mathrm{c_{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{ppo_2}})^+$) which is not a violation on A_1 (*i.e.* $\mathsf{acyclic}\left((\xrightarrow{\sigma} \cap \overset{\mathrm{c_{drf}}}{\leftrightarrow}) \cup \xrightarrow{\mathrm{ppo_1}}\right)$). A *minimal violation* is a violation cycle which is minimal. We write $A_1 \,.\, \mathsf{mv}_{\mathsf{A_2}}(E, \xrightarrow{\sigma})$ when $\xrightarrow{\sigma}$

is a minimal violation of A_2 *w.r.t.* A_1 (we omit A_1 and A_2 when they are clear from the context). Formally, we have:

Definition 70 (Minimal violation of A_2 *w.r.t.* A_1)

$$\begin{gathered} A_1 \,.\, \mathsf{mv}_{\mathsf{A_2}}(E, \xrightarrow{\sigma}) \triangleq \mathsf{cycle}(\xrightarrow{\sigma}) \wedge \left(\xrightarrow{\sigma} \subseteq (\overset{\text{c}_{\text{drf}}}{\leftrightarrow} \cup \xrightarrow{\text{ppo}_2})^+\right) \wedge \\ \mathsf{acyclic}\left((\xrightarrow{\sigma} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow}) \cup \xrightarrow{\text{ppo}_1}\right) \wedge \\ \neg\left(\exists \xrightarrow{\sigma}{}', (\xrightarrow{\sigma}{}' \neq \xrightarrow{\sigma}) \wedge (\xrightarrow{\sigma}{}' \subseteq \xrightarrow{\sigma}) \wedge \mathsf{cycle}(\xrightarrow{\sigma}{}')\right) \end{gathered}$$

Consider the execution of the **iriw** test we give in Fig. 7.1: there is a cycle $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fre}} (f) \xrightarrow{\text{rfe}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fre}} (e) \xrightarrow{\text{po}} (a)$, hence a cycle in $\overset{\text{c}_{\text{drf}}}{\leftrightarrow} \cup \xrightarrow{\text{po}}$. Therefore, this execution exhibits a violation of SC *w.r.t.* Power for example. The execution of **iriw** given in Fig. 10.11 also exhibits a violation of SC *w.r.t.* Power: $(a) \xrightarrow{\text{po}} (b) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (f) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (c) \xrightarrow{\text{po}} (d) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (e) \xrightarrow{\text{po}} (a)$. Yet, this execution is valid on SC, whereas the execution of Fig. 7.1 is not. Hence the presence of a violation of A_2 *w.r.t.* A_1 in a given event structure is not sufficient to determine the validity of an associated execution on A_2.

To address this issue, we define the *oriented violations*. Given an execution (E, X), we write $\xrightarrow{\text{oc}_{\text{drf}}}$ for the oriented version of $\overset{\text{c}_{\text{drf}}}{\leftrightarrow}$, *i.e.* $x \xrightarrow{\text{oc}_{\text{drf}}} y \triangleq (x \xrightarrow{\text{com}} y) \wedge (\mathsf{proc}(x) \neq \mathsf{proc}(y))$ or equivalently $(x, y) \in \mathsf{ws}(X) \cup \mathsf{rf}(X) \cup \mathsf{fr}(X) \wedge (\mathsf{proc}(x) \neq \mathsf{proc}(y))$. We consider a cycle to be an oriented violation of A_2 *w.r.t.* A_1 when it is a cycle in $\xrightarrow{\text{oc}_{\text{drf}}} \cup \xrightarrow{\text{ppo}_2}+$ (*i.e.* $\xrightarrow{\sigma} \subseteq (\xrightarrow{\text{oc}_{\text{drf}}} \cup \xrightarrow{\text{ppo}_2})^+$) which is not an oriented violation on A_1 (*i.e.* $\mathsf{acyclic}\left((\xrightarrow{\sigma} \cap \xrightarrow{\text{oc}_{\text{drf}}}) \cup \xrightarrow{\text{ppo}_1}\right)$). We write $\mathsf{mov}(E, X, \xrightarrow{\sigma})$ when $\xrightarrow{\sigma}$ is a minimal oriented violation:

Definition 71 (Minimal oriented violation of A_2 *w.r.t.* A_1)

$$\begin{gathered} A_1 \,.\, \mathsf{mov}_{\mathsf{A_2}}(E, \xrightarrow{\sigma}) \triangleq \mathsf{cycle}(\xrightarrow{\sigma}) \wedge \left(\xrightarrow{\sigma} \subseteq (\xrightarrow{\text{oc}_{\text{drf}}} \cup \xrightarrow{\text{ppo}_2})^+\right) \wedge \\ \mathsf{acyclic}\left((\xrightarrow{\sigma} \cap \xrightarrow{\text{oc}_{\text{drf}}}) \cup \xrightarrow{\text{ppo}_1}\right) \wedge \\ \neg\left(\exists \xrightarrow{\sigma}{}', (\xrightarrow{\sigma}{}' \neq \xrightarrow{\sigma}) \wedge (\xrightarrow{\sigma}{}' \subseteq \xrightarrow{\sigma}) \wedge \mathsf{cycle}(\xrightarrow{\sigma}{}')\right) \end{gathered}$$

For example, the cycle in Fig. 10.11 is not an oriented violation, whereas the cycle in Fig. 7.1 is. Thus, a minimal oriented violation is in particular a minimal violation, but the converse is not true.

Indeed, a minimal oriented violation is defined *w.r.t.* an execution witness, whereas a minimal violation is independent of any execution witness. This allows us to give in Thm. 12 a characterisation of all the stable executions associated with an event structure by reasoning on the event structure only. Hence, we spare the cost of enumerating all the execution witnesses to check their stability.

11.1.2 Covering the minimal violations

We show in the following that, given a conflict relation $\xrightarrow{\mathrm{c}}$ and a synchronisation relation $\xrightarrow{\mathrm{s}}$, it is enough to synchronise by $\xrightarrow{\mathrm{s}}$ the conflicting accesses of $\xrightarrow{\mathrm{c}}$ that belong to a minimal violation.

We call these conflicts *minimal conflicts*. Given an event structure E, we write $\xrightarrow{\mathrm{c_{min}}}$ for the minimal conflicts of E, *i.e.* $\xrightarrow{\mathrm{c}}$ restricted to the minimal violations of E:

Definition 72 (Minimal conflicts)

$$m_1 \xrightarrow{\mathrm{c_{min}}} m_2 \triangleq m_1 \xrightarrow{\mathrm{c}} m_2 \wedge (\exists \xrightarrow{\sigma}, \mathsf{mv}(E, \xrightarrow{\sigma}) \wedge m_1 \xrightarrow{\sigma} m_2)$$

We first show that if all the conflicts of the minimal violations (of A_2 $w.r.t.$ A_1) of an execution (E, X) valid on A_1 are covered, there cannot be any minimal oriented violations in (E, X). A minimal oriented violation is what makes an execution invalid on A_2. Hence forbidding them ensures that (E, X) is valid on A_2 as well. This means that synchronising the conflicting accesses of the minimal violations of an execution ensures its stability:

Theorem 11 (Covering minimal cycles)

$$\forall \xrightarrow{\mathrm{c}} \xrightarrow{\mathrm{s}}, \Big((\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) = (\xleftrightarrow{\mathrm{c_{drf}}}, \xrightarrow{\mathrm{s_{drf}}}) \vee (\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) = (\xrightarrow{\mathrm{c_{lf}}}, \xrightarrow{\mathrm{s_{lf}}}) \Big) \Rightarrow \\ \mathsf{covering}(\xrightarrow{\mathrm{c_{min}}}, \xrightarrow{\mathrm{s}})$$

Proof *Let (E, X) be an execution valid on A_1 where $\xrightarrow{\mathrm{c_{min}}}$ is arbitrated by $\xrightarrow{\mathrm{s}}$. Consider by contradiction a cycle in $A_2 . \mathsf{ghb}(E, X)$. Hence, there exists a minimal oriented violation $\xrightarrow{\sigma}$ of A_2 w.r.t. A_1 in (E, X).*

We know that $\xrightarrow{\sigma}$ is a cycle in $\xrightarrow{\mathrm{oc_{drf}}} \cup \xrightarrow{\mathrm{ppo_2}}{+}$. Therefore, we know that $\xrightarrow{\sigma}$ is equal to $(\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{oc_{drf}}}) \cup (\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{ppo_2}})$. By hypothesis, since $\xrightarrow{\sigma}$ is a minimal oriented violation, all the conflicts in $\xrightarrow{\sigma}$ are arbitrated by $\xrightarrow{\mathrm{s}}$.

- *Suppose $(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) = (\xleftrightarrow{\mathrm{c_{drf}}}, \xrightarrow{\mathrm{s_{drf}}})$. Since $\xrightarrow{\mathrm{s_{drf}}}$ is compatible with $\xrightarrow{\mathrm{com}}$, all the pairs in $\xrightarrow{\mathrm{oc_{drf}}}$ are in $\xrightarrow{\mathrm{s_{drf}}}$. Hence $\xrightarrow{\sigma} = (\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{s_{drf}}}) \cup (\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{ppo_2}})$. Since $\xrightarrow{\mathrm{ppo_2}} \subseteq \xrightarrow{\mathrm{po}}$, we have $\xrightarrow{\sigma} \subseteq (\xrightarrow{\mathrm{s_{drf}}} \cup \xrightarrow{\mathrm{po}})^{+}$. Since $\xrightarrow{\mathrm{s_{drf}}}$ is compatible with $\xrightarrow{\mathrm{po}}$, there cannot be any such cycle $\xrightarrow{\sigma}$.*

- *Suppose $(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) = (\xrightarrow{\mathrm{c_{lf}}}, \xrightarrow{\mathrm{s_{lf}}})$. Since $\xrightarrow{\sigma}$ is a cycle in $(\xrightarrow{\mathrm{oc_{drf}}} \cup \xrightarrow{\mathrm{ppo_2}})^{+}$, we know that $\xrightarrow{\sigma}$ is included in $(\xrightarrow{\mathrm{ws}} \cup \xrightarrow{\mathrm{fr}} \cup \xrightarrow{\mathrm{rf}} \cup \xrightarrow{\mathrm{ppo_2}})^{+}$. Since $\xrightarrow{\mathrm{ws}}$ and $\xrightarrow{\mathrm{fr}}$ are in $\xrightarrow{\mathrm{ghb_1}}$, $\xrightarrow{\sigma}$ is included in $(\xrightarrow{\mathrm{ghb_1}} \cup \xrightarrow{\mathrm{ppo_2}\backslash 1} \cup (\xrightarrow{\mathrm{grf_2}\backslash 1}; \xrightarrow{\mathrm{ppo_2}}))^{+}$, i.e. in $(\xrightarrow{\mathrm{ghb_1}} \cup \xrightarrow{\mathrm{c_{lf}}})^{+}$. Since $\xrightarrow{\mathrm{s_{lf}}}$ covers $\xrightarrow{\mathrm{c_{lf}}}$ and is compatible with $\xrightarrow{\mathrm{grf_2}}$ and $\xrightarrow{\mathrm{ppo_2}}$, $\xrightarrow{\sigma}$ is included in $(\xrightarrow{\mathrm{ghb_1}} \cup \xrightarrow{\mathrm{s_{lf}}})^{+}$. Since $\xrightarrow{\mathrm{s_{lf}}}$ is compatible with $\xrightarrow{\mathrm{ghb_1}}$, there cannot be any such cycle $\xrightarrow{\sigma}$.* □

11.1.3 Critical cycles

D. Shasha and M. Snir provide in [SS] an analysis to place barriers in a program, in order to enforce a SC behaviour. They examine the *critical*

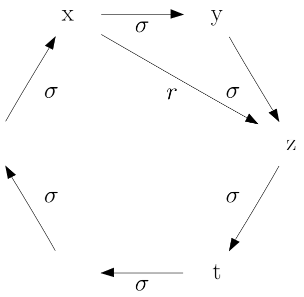


Figure 11.1: A Chord in a Cycle

cycles of an execution, and show that placing a barrier along each program order arrow of such a cycle (each *delay* arrow) is enough to restore SC. However, this work does not provide any semantics of weak memory models, hence does not address memory models with store buffering, such as TSO or x86, or the ones that relax store atomicity, such as Power. We generalise this approach to the models embraced by our framework.

We adapt here the definition of critical cycles of [SS], and the characterisation for finding these critical cycles defined in [SS] to our framework. We show that the minimal cycles are the critical cycles, hence this characterisation allows us to find the minimal cycles.

Intuitively, a cycle is minimal if there is not shortcut in it that could lead to a smaller cycle. Consider the cycle depicted in Fig. 11.1. The relation $\xrightarrow{\sigma}$ forms a cycle. Yet, it is not minimal, since there is a shortcut $\xrightarrow{\text{r}}$ between x and z.

The notion of shortcut is formalised by the notion of *chord*. Given a cycle $\xrightarrow{\sigma}$ and a relation $\xrightarrow{\text{r}}$, we say that there is no chord in $\xrightarrow{\sigma}$ *via* $\xrightarrow{\text{r}}$ when there is no shortcut in $\xrightarrow{\sigma}$ *via* $\xrightarrow{\text{r}}$, *i.e.*:

Definition 73 (Absence of chord in $\xrightarrow{\sigma}$ *via* $\xrightarrow{\text{r}}$)

$$\mathsf{no\text{-}chord}(\xrightarrow{\sigma}, \xrightarrow{\text{r}}) \triangleq \forall xyzt, x \xrightarrow{\sigma} y \wedge z \xrightarrow{\sigma} t \wedge x \xrightarrow{\text{r}} z \Rightarrow y = z$$

We define a critical cycle $\xrightarrow{\sigma}$ as a cycle in $(\overset{\text{cdrf}}{\leftrightarrow} \cup \xrightarrow{\text{ppo}_2})^+$, such that:

- there is no chord in $\xrightarrow{\sigma}$ *via* $(\xrightarrow{\text{ppo}_2})^+$ nor *via* $(\overset{\text{cdrf}}{\leftrightarrow})^+$, and
- $\xrightarrow{\sigma}$ contains at least one occurrence of $\xrightarrow{\text{ppo}_{2\setminus 1}}$, and
- $(\xrightarrow{\sigma} \cap \overset{\text{cdrf}}{\leftrightarrow})$ is acyclic, and
- for all access x in $\xrightarrow{\sigma}$, there exists at least another access y in $\xrightarrow{\sigma}$ competing with x, *i.e.* $(x, y) \in (\xrightarrow{\sigma} \cap \overset{\text{cdrf}}{\leftrightarrow})$.

Formally, we have (writing $\mathsf{critical}(\xrightarrow{\sigma})$ when $\xrightarrow{\sigma}$ is a critical cycle):

Definition 74 (Critical Cycle)

$$\begin{aligned}
\mathsf{critical}(E, \xrightarrow{\sigma}) \quad \triangleq \quad & \mathsf{cycle}(\xrightarrow{\sigma}) \wedge \left(\xrightarrow{\sigma} \subseteq (\overset{\mathrm{c_{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{ppo}_2})^{+}\right) \wedge \\
& \mathsf{no\text{-}chord}(\xrightarrow{\sigma}, \xrightarrow{\mathrm{ppo}_2}) \wedge \mathsf{no\text{-}chord}(\xrightarrow{\sigma}, \overset{\mathrm{c_{drf}}}{\leftrightarrow}) \wedge \\
& \exists x \neq y, (x, y) \in (\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{ppo}_{2\setminus 1}}) \wedge \mathsf{acyclic}(\xrightarrow{\sigma} \cap \overset{\mathrm{c_{drf}}}{\leftrightarrow}) \wedge \\
& \forall m_1 \in \xrightarrow{\sigma}, \exists m_2, m_2 \in \xrightarrow{\sigma} \wedge m_1 \overset{\mathrm{c_{drf}}}{\leftrightarrow} m_2
\end{aligned}$$

For example, consider the execution of **iriw** in Fig. 10.11. The cycle $\xrightarrow{\sigma}$ exhibited by this execution is a critical cycle of SC (*i.e.* a minimal violation of SC). Indeed, this is a cycle in $(\overset{\mathrm{c_{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{po}})$ which restriction to $\overset{\mathrm{c_{drf}}}{\leftrightarrow}$ is acyclic. There is no chord in $(\xrightarrow{\sigma})^{+}$ *via* $(\overset{\mathrm{c_{drf}}}{\leftrightarrow})^{+}$ nor *via* $\xrightarrow{\mathrm{po}}$. Moreover, there is one occurrence of $\xrightarrow{\mathrm{po}}$, *e.g.* $(a) \xrightarrow{\mathrm{po}} (b)$, and each access in $\xrightarrow{\sigma}$ is competing with another one, *e.g.* $(f) \overset{\mathrm{c_{drf}}}{\leftrightarrow} (c)$.

We show formally that a cycle is critical if and only if it is minimal.

Lemma 18 (A critical cycle is a minimal violation)

$$\forall E \xrightarrow{\sigma}, \mathsf{critical}(E, \xrightarrow{\sigma}) \Rightarrow \mathsf{mv}(E, \xrightarrow{\sigma})$$

Proof *Let E be an event structure and $\xrightarrow{\sigma}$ be a critical cycle for E. We know by hypothesis that $\xrightarrow{\sigma}$ is a cycle in $(\overset{\mathrm{c_{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{ppo}_2})^{+}$.*

Let us show that $\mathsf{acyclic}((\xrightarrow{\sigma} \cap \overset{\mathrm{c_{drf}}}{\leftrightarrow}) \cup \xrightarrow{\mathrm{ppo}_1})$*. Suppose by contradiction that there is a cycle in $((\xrightarrow{\sigma} \cap \overset{\mathrm{c_{drf}}}{\leftrightarrow}) \cup \xrightarrow{\mathrm{ppo}_1})$. This cycle is either $\xrightarrow{\sigma}$ or a cycle included in $\xrightarrow{\sigma}$. Since there cannot be any chords in $\xrightarrow{\sigma}$* via $(\xrightarrow{\mathrm{ppo}_2})^{+}$ *nor* via $\overset{\mathrm{c_{drf}}}{\leftrightarrow}$*, this cycle cannot be included in $\xrightarrow{\sigma}$. Hence it is $\xrightarrow{\sigma}$; but since there is at least one occurrence of $\xrightarrow{\mathrm{ppo}_{2\setminus 1}}$ in $\xrightarrow{\sigma}$, this cycle cannot be $\xrightarrow{\sigma}$.*

Finally, there is no cycle $\xrightarrow{\sigma}{}'$ distinct of $\xrightarrow{\sigma}$ and included in $\xrightarrow{\sigma}$, otherwise there would be chords in $\xrightarrow{\sigma}$ via $(\xrightarrow{\mathrm{ppo}_2})^{+}$ *or* via $\overset{\mathrm{c_{drf}}}{\leftrightarrow}$.

Lemma 19 (A minimal violation is critical)

$$\forall E \xrightarrow{\sigma}, \mathsf{mv}(E, \xrightarrow{\sigma}) \Rightarrow \mathsf{critical}(E, \xrightarrow{\sigma})$$

Proof *Let E be an event structure and $\xrightarrow{\sigma}$ be a minimal violation for E. We know by hypothesis that $\xrightarrow{\sigma}$ is a cycle in $(\overset{\mathrm{c_{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{ppo}_2})^{+}$.*

There cannot be any chord in $\xrightarrow{\mathrm{ppo}_2}$ nor $\overset{\mathrm{c_{drf}}}{\leftrightarrow}$ in $\xrightarrow{\sigma}$, otherwise $\xrightarrow{\sigma}$ would not be minimal. For the same reason, we have $\mathsf{acyclic}(\xrightarrow{\sigma} \cap \overset{\mathrm{c_{drf}}}{\leftrightarrow})$.

There is at least one occurrence of $\xrightarrow{\mathrm{ppo}_{2\setminus 1}}$ in $\xrightarrow{\sigma}$, otherwise we would contradict the hypothesis that $\mathsf{acyclic}((\xrightarrow{\sigma} \cap \overset{\mathrm{c_{drf}}}{\leftrightarrow}) \cup \xrightarrow{\mathrm{ppo}_1})$.

Finally, consider an event m_1 in $\xrightarrow{\sigma}$, and suppose that there is no m_2 competing with m_1 in $\xrightarrow{\sigma}$. In particular, this means that all the arrows having m_1 as either source or target are $\xrightarrow{\mathrm{ppo}_2}$ arrows, otherwise we would trivially contradict our hypothesis that no event of $\xrightarrow{\sigma}$ competes with m_1. Hence there exist $x \xrightarrow{\sigma} y$ such that $x \xrightarrow{\mathrm{ppo}_2} m_1 \xrightarrow{\mathrm{ppo}_2} y$. But then we have a chord in $\xrightarrow{\mathrm{ppo}_2}$ in $\xrightarrow{\sigma}$, since $(x, y) \in (\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{ppo}_2})$.

11.1.4 A characterisation of the minimal cycles

The critical cycles are the minimal cycles, hence it is enough to adapt the characterisation of [SS] to our framework to find the minimal cycles of a given event structure. Consider a pair (x, y) in a minimal cycle $\xrightarrow{\sigma}$. The characterisation of [SS] adapted to our framework ensures that if $\mathsf{proc}(x) = \mathsf{proc}(y)$, then:

- $x \xrightarrow{\text{ppo}_2} y$ (since $\xrightarrow{\sigma} \subseteq (\overset{\text{cdrf}}{\leftrightarrow} \cup \xrightarrow{\text{ppo}_2})^+$),
- and there is no other access z in $\xrightarrow{\sigma}$ such that $(x, z) \in (\xrightarrow{\sigma} \cap \xrightarrow{\text{ppo}_2})$ and $(z, y) \in (\xrightarrow{\sigma} \cap \xrightarrow{\text{ppo}_2})$ (otherwise there would be a chord in $\xrightarrow{\sigma}$ *via* $(\xrightarrow{\text{ppo}_2})^+$),
- and $\mathsf{loc}(x) \neq \mathsf{loc}(y)$ (otherwise there would be a chord in $\xrightarrow{\sigma}$ *via* $(\overset{\text{cdrf}}{\leftrightarrow})^+$).

For a pair $x \xrightarrow{\sigma} y$ such that $\mathsf{proc}(x) \neq \mathsf{proc}(y)$, the characterisation ensures that x and y have the same location, and either:

- at least one of them is a write (*i.e.* $x \overset{\text{cdrf}}{\leftrightarrow} y$), and there is no other access z such that $(x, z) \in ((\xrightarrow{\sigma})^+ \cap (\overset{\text{cdrf}}{\leftrightarrow})^+)$ or $(z, y) \in ((\xrightarrow{\sigma})^+ \cap (\overset{\text{cdrf}}{\leftrightarrow})^+)$ (otherwise there would be a chord in $\xrightarrow{\sigma}$ *via* $(\overset{\text{cdrf}}{\leftrightarrow})^+$);
- or x and y are both reads, and there exists a unique write z in $\xrightarrow{\sigma}$, such that $(x, z) \in ((\xrightarrow{\sigma})^+ \cap (\overset{\text{cdrf}}{\leftrightarrow})^+)$ and $(z, y) \in ((\xrightarrow{\sigma})^+ \cap (\overset{\text{cdrf}}{\leftrightarrow})^+)$. The unicity of z is ensured by the absence of chord in $\xrightarrow{\sigma}$ *via* $(\overset{\text{cdrf}}{\leftrightarrow})^+$.

Hence a minimal cycle is a cycle in $(\overset{\text{cdrf}}{\leftrightarrow} \cup \xrightarrow{\text{ppo}_2})^+$, such that:

- per processor, there is only one pair (x, y) of accesses on this processor, such that $x \xrightarrow{\text{ppo}_2} y$ and $\mathsf{loc}(x) \neq \mathsf{loc}(y)$, and
- per memory location, there are at most three accesses relative to this location, and these accesses are from distinct processors. For a given location x, the only possible configurations are (Wx, Wx), (Wx, Rx), (Rx, Wx) or (Rx, Wx, Rx).

Consider again the cycle exhibited by the execution of **iriw** in Fig. 10.11. This cycle would be captured by our characterisation. Indeed the only accesses on the same processor are $(a) \xrightarrow{\text{po}} (b)$ on P_0 and $(c) \xrightarrow{\text{po}} (d)$ on P_1: these events are relative to distinct locations, and there is no access in between them in $\xrightarrow{\text{po}}$. Moreover, for each memory location involved, there are at most three accesses relative to this location, all of them being from distinct processors, and one of them at least is a write; *e.g.* for x there are the read (a) on P_0, the read (d) on P_1 and the write (e) on P_2.

This gives a simple characterisation for computing minimal cycles.

11.2 Stability from any architecture to SC

We show here our final result: we give a characterisation of the executions (E, X) that are stable from any architecture A to SC (*i.e.* we consider here that $\mathsf{mv}(E, \xrightarrow{\sigma}) \triangleq A.\,\mathsf{mv}_{SC}(E, \xrightarrow{\sigma})$ and that $\mathsf{mov}(E, X, \xrightarrow{\sigma}) \triangleq A.\,\mathsf{mov}_{SC}(E, X, \xrightarrow{\sigma})$).

We show indeed that an execution is stable from A to SC if and only if there is no minimal violation of SC *w.r.t.* A in E.

The intuition for the direct way is the following. Let (E, X) be an execution valid on A. Suppose by contradiction that there is a minimal violation of SC *w.r.t.* A in E, *e.g.* the execution of the **iriw** test given in Fig. 10.11. This execution exhibits a minimal violation of SC *w.r.t.* Power, *i.e.* $(a) \xrightarrow{\text{po}} (b) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (f) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (c) \xrightarrow{\text{po}} (d) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (e) \overset{\text{c}_{\text{drf}}}{\leftrightarrow} (a)$. Again, a minimal violation does not forbid the execution, and indeed the execution of Fig. 10.11 is authorised on SC (since there is no cycle in $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po}}$).

Yet, even if a minimal violation of SC is not necessarily an actual violation of SC (*i.e.* a cycle in $\xrightarrow{\text{ghb}_2}$), it is possible to build an execution Y (by transforming X) associated with the same event structure, which contains an actual violation of SC. In Fig. 10.11, if the $\xrightarrow{\text{rf}}$ relations $(f) \xrightarrow{\text{rf}} (b)$ and $(e) \xrightarrow{\text{rf}} (d)$ become $\xrightarrow{\text{fr}}$ ones (*i.e.* $(b) \xrightarrow{\text{fr}} (f)$ and $(d) \xrightarrow{\text{fr}} (e)$), then we build the execution of **iriw** depicted in Fig. 7.1. This execution is forbidden on SC, because it contains a minimal oriented violation: $(a) \xrightarrow{\text{po}} (b) \xrightarrow{\text{fr}} (f) \xrightarrow{\text{rf}} (c) \xrightarrow{\text{po}} (d) \xrightarrow{\text{fr}} (e) \xrightarrow{\text{rf}} (a)$. Hence we build an execution violating SC (the one in Fig. 7.1), from an execution which does not violate SC, but exhibits a minimal violation of SC (the one in Fig. 10.11).

We formalise this idea in the following lemma. It states that for any execution (E, X) valid on A such that E contains a minimal violation cycle $\xrightarrow{\sigma}$ (such as the one of Fig. 10.11), we can build another execution Y, valid on A and associated with E as well, in which $\xrightarrow{\sigma}$ is a minimal oriented violation (such as the one of Fig. 7.1):

Lemma 20 (Existence of an execution with oriented violation)

$$\begin{gathered}\forall E X \xrightarrow{\sigma}, A.\,\mathsf{valid}(E, X) \wedge \mathsf{mv}(E, \xrightarrow{\sigma}) \Rightarrow \\ (\exists Y, A.\,\mathsf{valid}(E, Y) \wedge \mathsf{mov}(E, Y, \xrightarrow{\sigma}))\end{gathered}$$

Proof *Let (E, X) be an execution valid on A. Let $\xrightarrow{\sigma}$ be a minimal violation of SC w.r.t. A in E. By definition of violation, we know that* $\mathsf{acyclic}((\xrightarrow{\sigma} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow}) \cup \xrightarrow{\text{ppo}_A})$. *Hence we know by Lem. 8 that we can build an execution Y associated with E and valid on A from any linear extension $\xrightarrow{\text{le}}$ of the order $((\xrightarrow{\sigma} \cap \overset{\text{c}_{\text{drf}}}{\leftrightarrow}) \cup \xrightarrow{\text{ppo}_A})^+$.*

Let us show that $\xrightarrow{\sigma}$ is a minimal oriented violation of SC w.r.t. A in (E, Y). Since $\xrightarrow{\sigma}$ is a minimal cycle in E, we know it is a minimal cycle in (E, Y). Since the strong architecture is SC here, we have $\xrightarrow{\text{ppo}_2} = \xrightarrow{\text{po}}$. Hence, we need to show that $\xrightarrow{\sigma}$ is included in $(\xrightarrow{\text{oc}_{\text{drf}}} \cup \xrightarrow{\text{po}})^+$ in (E, Y). Thus, for all x and y such that $x \xrightarrow{\sigma} y$, we need to *show that* $(x, y) \in (\xrightarrow{\mathrm{r}})^{+}$*, where* $m_1 \xrightarrow{\mathrm{r}} m_2 \triangleq ((m_1, m_2) \in (\mathsf{ws}(Y) \cup \mathsf{rf}(Y) \cup \mathsf{fr}(Y)) \wedge (\mathsf{proc}(m_1) \neq \mathsf{proc}(m_2))$ $(m_1 \xrightarrow{\mathrm{po}} m_2)$.

Consider two events $x \xrightarrow{\sigma} y$*. Since* $\xrightarrow{\sigma}$ *is a violation of SC, we have* $(x, y) \in (\overset{\mathrm{c}_{\mathrm{drf}}}{\leftrightarrow} \cup \xrightarrow{\mathrm{po}})^{+}$*. Let us reason by induction over this statement. In the base case, let us do a case disjunction.*

- *When* $(x, y) \in \overset{\mathrm{c}_{\mathrm{drf}}}{\leftrightarrow}$*, we do case disjunction over the directions of* x *and* y*. Since they are in* $\overset{\mathrm{c}_{\mathrm{drf}}}{\leftrightarrow}$*, we know that they cannot be both reads.*
  - *If* x *and* y *are both writes, we know since they are in* $\overset{\mathrm{c}_{\mathrm{drf}}}{\leftrightarrow}$ *that they are to the same location and from distinct processors. Moreover, we know that* $x \xrightarrow{\sigma} y$*. Therefore by definition of extracted write serialisation (see Lem. 8), they are in* $\mathsf{ws}(\xrightarrow{\mathrm{le}})$*, i.e. in* $\mathsf{ws}(Y)$*.*
  - *If* x *is a read and* y *a write, we know since* (E, Y) *is valid on* A *that there exists a write* w_x *such that* $(w_x, x) \in \mathsf{rf}(Y)$*. Hence, by definition of* $\mathsf{rf}(Y)$*, we know that* $w_x \xrightarrow{\mathrm{le}} x$*. Moreover, we know that* $(x, y) \in (\xrightarrow{\sigma} \cap \overset{\mathrm{c}_{\mathrm{drf}}}{\leftrightarrow})$ *by hypothesis, hence* $x \xrightarrow{\mathrm{le}} y$*. Thus by transitivity we have* $w_x \xrightarrow{\mathrm{le}} y$*. Since* w_x *and* y *are both writes, and to the same location, we know by definition that* $(w_x, y) \in \mathsf{ws}(Y)$*. Hence, since* $(w_x, x) \in \mathsf{rf}(Y)$ *and* $(w_x, y) \in \mathsf{ws}(Y)$*, we have* $(x, y) \in \mathsf{fr}(Y)$*.*
  - *If* x *is a write and* y *a read, we know since* (E, Y) *is valid on* A *that there exists a write* w_y *such that* $(w_y, y) \in \mathsf{rf}(Y)$*. Suppose* $x = w_y$*; in this case, we have* $(x, y) \in \mathsf{rf}(Y)$*, hence the result. Suppose* $x \neq w_y$*. In this case, since* x *and* w_y *are both writes to the same location, we have* $(x, w_y) \in \mathsf{ws}(Y) \vee (w_y, x) \in \mathsf{ws}(Y)$*.*
    - *When* $(x, w_y) \in \mathsf{ws}(Y)$*, suppose* $\mathsf{proc}(x) = \mathsf{proc}(w_y)$*. In this case, we know that* $x \xrightarrow{\mathrm{po}} w_y$ *(since in* $\xrightarrow{\mathrm{ws}}$ *and from the same processor). Otherwise (i.e. if* $\mathsf{proc}(x) \neq \mathsf{proc}(w_y)$*),* (x, w_y) *is in* $\mathsf{ws}(Y)$ *and the two events are from distinct processors. The same reasoning applies to* (w_y, y)*: we have either* $w_y \xrightarrow{\mathrm{po}} y$ *if they are on the same processor, or* $(w_y, y) \in \mathsf{rf}(Y)$ *if not.*
    - *When* $(w_y, x) \in \mathsf{ws}(Y)$*, since* $(w_y, y) \in \mathsf{rf}(Y)$*, we know that* w_y *is the maximal previous write to* $\mathsf{loc}(y)$ *before* y *in* $\xrightarrow{\mathrm{le}}$*. Since* $(x, y) \in (\xrightarrow{\sigma} \cap \overset{\mathrm{c}_{\mathrm{drf}}}{\leftrightarrow})$ *by hypothesis, we have* $x \xrightarrow{\mathrm{le}} y$*. Since* $(w_y, x) \in \mathsf{ws}(Y)$*, we have* $w_y \xrightarrow{\mathrm{le}} x$*. Hence* x *occurs in between* w_y *and* y *in* $\xrightarrow{\mathrm{le}}$*, a contradiction.*
- *When* $x \xrightarrow{\mathrm{po}} y$*, we trivially have* $(x, y) \in ((\xrightarrow{\sigma} \cap \xrightarrow{\mathrm{oc}_{\mathrm{drf}}}) \cup \xrightarrow{\mathrm{po}})^{+}$*.*

The transitive case follows by immediate induction.

Finally, we show that an execution (E, X) is stable from A to SC if and only if there is no minimal violation of SC *w.r.t.* A in E:

Theorem 12 (Characterisation of stability from A to SC)

$$\forall E, (\forall X, \mathsf{stable}_{\mathsf{A,SC}}(E, X)) \Leftrightarrow \neg(\exists \xrightarrow{\sigma}, \mathsf{mv}(E, \xrightarrow{\sigma}))$$

Proof *Let E be an event structure.*

- $\Rightarrow$ *Let X be an associated execution witness; (E, X) is stable by hypothesis. Suppose by contradiction the existence of a minimal violation $\xrightarrow{\sigma}$ of SC w.r.t. A. In this case, by Lem. 20, we know that there exists another execution witness Y, such that (E, Y) is valid on A, in which $\xrightarrow{\sigma}$ is a minimal oriented violation of SC w.r.t. A. Since all the executions associated with E are stable, (E, Y) is stable. Since (E, Y) is valid on A and stable, we know by definition of stable that (E, Y) is valid on SC. But since (E, Y) contains a minimal oriented violation of SC w.r.t. A, (E, Y) cannot be valid on SC.*

- $\Leftarrow$ *Suppose $(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) = (\overset{\mathrm{c_{drf}}}{\leftrightarrow}, \xrightarrow{\mathrm{s_{drf}}})$ or $(\xrightarrow{\mathrm{c}}, \xrightarrow{\mathrm{s}}) = (\overset{\mathrm{c_{lf}}}{\leftrightarrow}, \xrightarrow{\mathrm{s_{lf}}})$. Let (E, X) be an associated execution valid on A. We know that there is always such a X since any execution valid on SC is valid on a weaker architecture by Thm. 1: take for instance any linear extension of $\xrightarrow{\mathrm{po}}$. Suppose there is no minimal violation of SC w.r.t. A in E. Thus, $\xrightarrow{\mathrm{c_{min}}}$ is empty, hence we know that $\xrightarrow{\mathrm{c_{min}}}$ is trivially covered by $\xrightarrow{\mathrm{s}}$ in (E, X). Moreover, $\xrightarrow{\mathrm{s}}$ is covering for $\xrightarrow{\mathrm{c_{min}}}$ by Thm. 11. Therefore, we know by definition of covering that (E, X) is valid on SC.* □

This criterion means that to ensure the stability of the executions associated with an event structure E, we only need to check that E contains no minimal violation cycle. Since we gave a simple characterisation to compute the minimal cycles of a given event structure, we believe this criterion to be practicable.

Chapter 12

Related Work

DRF guarantee The DRF guarantee was initially proposed by S. Adve and M. Hill [AH, Adv93], as a simple model for programmers. It has been extensively studied since then, in high- and low-level models [BA, BPa, FFS, BPb]. V. Saraswat *et al.* [SJMvP] call it the "fundamental property" of such models: they consider that a weak memory model should provide at least this guarantee, otherwise it is unpracticable. This is a common requirement amongst programmers dealing with weak memory models: *correctly synchronised* programs should provide a strong semantics [MPA]. Providing such a requirement without impairing the performance is a major difficulty [AB].

Barriers placement Synchronisation using locks is costly indeed, and often not scalable [FH]. To spare this cost, lock-free and *wait-free* synchronisation techniques were developed [Her], but they do not provide guarantees as strong as the DRF guarantee. Hence programs using lock-free techniques are often hard to understand and to debug in the context of weak memory models, in particular $w.r.t.$ the placement of the barriers used in the RMW primitives.

The work of D. Shasha and M. Snir [SS] provide an analysis to place barriers in a program, in order to enforce a SC behaviour. We have already discussed this approach in Sec. 11.1.3, which we generalise to the models embraced by our framework, and to the synchronisations we studied.

Recently, tools have been developed to check the placement of barriers in a given piece of code. The Checkfence tool [BAM] for example exhaustively checks the executions of a given program on a weak memory model, and verifies that all these executions are observationally equivalent to sequential ones. If so, there is no need for barriers. However, this tool does not apply to models with features à la Power, such as the lack of store atomicity and the cumulativity of barriers. Moreover, even if we give an over-approximation of the executions, we show that there is no need to check all the executions, but only the strong ones. We believe that our approach could be refined to a precise simulation of the executions, hence help improving such a tool.

Stability Several recent works have addressed the question of whether two models can be distinguished or not. For example, S. Burckhardt and M. Musuvathi examine in [BMa] whether a program running on a TSO machine can be simulated by enumerating only its SC executions. They distinguish a certain class of programs, the *TSO-safe* ones, which have this property. We believe that these programs are an instance of our stable ones, *i.e.* the stable programs from TSO to SC. They develop the notion of *borderline execution* and show that a program is TSO-safe if and only if it has no borderline execution. We believe that the notion of borderline execution is closely related to our minimal cycles. Yet, our characterisation of the stable programs in the general case is, to the best of our knowledge, a novel contribution.

Finally, S. Mador-Haim *et al.* [MHAM] have developed a tool that examines whether two memory models can be distinguished. The tool systematically computes tests from cycles, and run them against the models. We believe that our work could benefit to the design of such a tool, by making the computation of tests more efficient. Indeed we prove that an architecture is exactly distinguished from SC by the minimal violations, hence to distinguish a model from SC, we know that the tests to be run are exactly those corresponding to the minimal violations. The same optimisation applies to our diy testing tool [AMSS], which exhibits the weakenings of SC exhibited by a given machine by running tests which proceed from cycles.

Perspectives Writing programs that run on weak memory models is hard, since such programs endure non-trivial relaxations, which may be inhibited by synchronisation. We provide a formal study of stability in such models, leading to a characterisation of the stable programs.

We believe that this work could have direct applications for program verification. Indeed, S. Burckhardt and M. Musuvathi have exposed in [BMa] the idea that a given verification problem for concurrent programs in the context of weak memory models could be solved in two steps. First, by solving the problem for SC *via* standard verification methodologies. Second, by proving that all the traces of the program are SC. In our terms, the second step amounts to proving the program to be stable.

Moreover, the efficient enumeration of executions is a well-known issue of weak memory models [HVM$^+$, BMa]. The existing tools address models with store buffering only (such as Sparc TSO). We believe that our work could help extending these methodologies to models that relax store atomicity. In addition, since a program that is not stable is arguably erroneous, we would only have to examine the stable programs, *i.e.* only the strong executions.

Hence, we believe that our work could help improving the scalability of these tools, by sparing them the cost of checking the additional non-determinism induced by weak memory models, as exposed in [BMa].

Finally, we believe that this work could be of interest to several other communities, in particular hardware architects, compiler writers and language designers.

From a hardware design point of view, we have highlighted some precise points in the semantics of Power's synchronisation primitives that could be relaxed. This could lead to optimisations, for example if the vendors give the programmers access to weaker barriers than the existing ones. Moreover, we have highlighted some lack of details in this documentation which worsen the difficulty of writing both correct and efficient programs, and which should be clarified to help programmers write concurrent programs.

From a compilation point of view, we believe that our work can help proving that *program transformations*, *e.g.* performed by compilers, are sound in the context of weak memory models. If a transformation of a given program is proved to be stable, then the transformation does not introduce additional, potentially erroneous behaviours. The correctness of program transformations has already been studied in the context of high-level memory models [Boea, Sev08], and recently for low-level ones, with a limited store atomicity relaxation [BMS]. These previous works examine the correctness of a transformation in a given model. Our work could lead to proofs of transformations from one model to another, via *backward compatibility* proofs, such as our stability ones.

From a language design point of view, we believe that our study could be a foundation for the design of high-level memory models. Following S. Adve and H.-J. Boehm in [AB], we believe indeed that software and hardware memory models should be *co-designed*. This means that the development of a hardware memory model should not be oblivious to the software it might interfere with, and conversely. Hence parallel languages should *enforce* a strong programming discipline, *e.g.* the DRF guarantee. We believe that our stability is a good candidate as a basis for such a discipline.

Part V

Conclusion and Perspectives

Chapter 13

Conclusion

We're Sgt. Pepper's Lonely
Hearts Club Band,
We hope you have enjoyed the
show,
Sgt. Pepper's Lonely Hearts
Club Band,
We're sorry but it's time to go.

*The Beatles—Sgt. Pepper's
Lonely Hearts Club Band
(Reprise) [LM67b]*

13.1 Divining Chicken Entrails

13.1.1 Reading the Documentations

As I mentioned several times, reading the documentations may be an incredible effort. See for example one of the reviews we had:

> *I remember spending quite a bit of time myself trying to formalize the PPC model, and in particular pondering the meaning of the informal prose in the PPC manuals, a process I recall being likened to "divining chicken entrails" by Doug Lea.*

And indeed, some documentations, especially the Power one, are vague and ambiguous. I strongly advocate the idea that the documentations should be written in a clear language, even if it is not mathematics. For example, the Power documentation contains confusing subjunctive and conditional forms [pow09, p. 654, last bullet]:

> *A load [...] by a processor or mechanism (P1) is performed with respect to any processor or mechanism (P2) when the value to be*

> *returned by the load or instruction fetch can no longer be changed by a store by P2. A store by P1 is performed with respect to P2 when a load by P2 from the location accessed by the store will return the value store (or a value stored subsequently). [...]*

In this excerpt, the existence of a "store by P2" (for the performance of a load definition) or a "load by P2" (for the performance of a store definition) is hypothetical. Hence, according to this definition, the performance of an access depends on the existence of another hypothetical access.

One can also find puzzling forms to a non-native speaker [pow09, p. 661, 1st col, 1st §]:

> *The same [as in Sec. 6.1.4.1] applies if* whether *the Store instruction is executed depends on a conditional Branch instruction that in turn depends on the value returned by a preceding Load instruction.*

The "if whether" expression seems to me rather ambiguous. I believe that a documentation should be accessible to non-native speakers, because they could be, as myself, unable to understand correctly subtle linguistics issues. A solution to such problems is to design *formal* models.

13.1.2 Abstract Models

I believe that there are two main reasons for the documentation being vague. The first one lies in the fact that the vendors do not want to reveal too much of the implementation's secrets; on the one hand because these are industrial secrets, and on the other hand because the vendors want to give themselves some liberty in the implementation. The second one lies in the fact that some vendors do not have enough time to design formal models.

Hence the memory model of a given architecture has to be loose. However, I disbelieve that this means it has to be vague. For example, Sun and Alpha provide formal and precise models, and because they provide *abstract models*, they precisely do not reveal much of the implementation.

13.1.3 Simple Formal Models

I believe that the specifications should be formal indeed, but intelligible to any programmer. Hence, I believe that relaxed memory models should be simple (if not intuitive). As pointed out by H.-J. Boehm [Boe07]:

> *If we want more programmers to be able to write reasonably correct multithreaded code, we need a consistent story that's easy to teach. Based on what I've seen so far, many programmers are hopelessly confused when it comes to threads, in large part because they've been taught rules that either don't make sense or*

> *are too complicated. I'm not yet convinced that's fixable with the non-SC approach.*

I do not know whether we should restrict the memory models to SC. It seems to me that TSO and PSO models are understandable, an observation already made by M. Hill in [Hil98]. However, I believe that memory models should not be much weaker than these models, mainly because the weaknesses exhibited by more relaxed models are highly counter intuitive, which leads to bugs due to a misunderstanding of the model.

13.1.4 The Preserved Program Order Quest

For example the notion of dependency, as in RMO or Power, seems to me quite subtle and debatable. Moreover, as pointed out by S. Burckhardt and M. Musuvathi [BMb], decidability and verification issues are increased by a preserved program order which is too weak. For example, in the case of RMO or Power, the fact that a load-store pair may not preserved in the program order leads to such problems. In recent work by M. F. Atig *et al.* [ABBM], the *state reachability* problem is shown to be undecidable for memory models that relax the read-read and read-write pairs.

In addition, I believe that programmers, when facing a very weak preserved program order, may be tempted to exploit its weakness to increase performance. But then again, this might lead to subtle concurrency bugs due to a misunderstanding of the specification.

13.1.5 Strong Programming Disciplines

A solution, suggested by S. Adve and H.-J. Boehm [AB], could be to impose a very strong programming discipline, such as in the C++ model, where racy programs do not have any semantics. Hence the programmers are forced to respect this discipline if they want to guarantee the behaviour of their programs.

I like that idea, because I believe that programmers should not be smarter than they want to be. But I believe that they should have some liberty in the design of their code if they want to. I believe that the following suggestion by M. Hill [Hil98] is a promising approach:

> *[...] one can provide a first-class SC implementation and add optional relaxed support.*

In the C++ model for example, programmers have access to *low-level atomics* that have a weaker semantics, yet provide some guarantee. Hence the programmers can perfectly well use these constructions to increase the performance of their code, provided that they are careful.

13.2 A Reading Frame For Weak Memory Models

13.2.1 A Common Prism

If memory models are simple, and accept more or less the same broad rules, the design of a common reading frame should be enough to formalise them. Thus, the programmers would only have to understand a few rules to make sense out of a given model, and adapt their code to a few parameters' variations, in the spirit of our generic framework.

I believe that this is a promising approach. Compiler programmers for example often have to understand several very different memory models: the high-level source one(s), and the several target ones. If such models can be broadly understood in the same terms, it could ease the writing of the non-trivial pieces of code that compilers are.

13.2.2 Tests as Specifications

Moreover, I believe that memory models should be definable in terms of a few litmus tests which would highlight their specificities. This is feasible if there is a common framework that covers many possible models, and if the variations between these models can be separated by testing. The main advantage that I see in that approach is that, once the main rules of the common framework are understood, it only takes a quick glance at a small set of tests to understand what the novelties of a new memory model are.

I do not know what exact characteristics a memory model should have in order to be definable in terms of a few tests. However, I am certain that it requires memory models to tend towards much more simplicity than several existing ones.

Chapter 14

Perspectives

> Sgt. Pepper's Lonely Hearts
> Club Band,
> We'd like to thank you once
> again,
> Sgt. Pepper's one and only
> lonely hearts club band,
> It's getting very near the end...
>
> *The Beatles—Sgt. Pepper's*
> *Lonely Hearts Club Band*
> *(Reprise) [LM67b]*

14.1 Automatisation

I would like to see if the stability results presented in Chap. 11 could lead to the design and writing of automatic tools. These could be inspired of S. Burckhardt's *et al.* [BAM, BMa], which are able to automatically insert fences in a piece of code [BAM], and to enumerate efficiently the executions of a given program on a weak memory model [BMa]. I believe that the ideas I presented in Chap. 11 are closely related to the ones used by S. Burckhardt's *et al.* , and that my results could help extending these tools to very weak models such as Power.

14.2 Formalisation of Diy

I would like to formalise the behaviour of our diy tool (see Chap. 7). In particular, I would like to see to what extent diy is complete *w.r.t.* our generic framework (see Chap. 3). I believe that our study of critical cycles (see Sec. 11.1) could be a promising lead towards this goal.

Moreover, I believe that it could be interesting to formalise our diy approach using ideas from *black box checking* [PVY99] or *adaptive model checking* [GPY06]. In these approaches, the structure of the system that is checked or tested is unknown, or not up to date. Therefore, one needs to iterate the testing or checking in order to learn more about the system, and feed the checker or the tester back with that information. I believe that such ideas are quite close from what we did with our Phat Experiment: we had an incomplete and incorrect model in mind, and without knowing the exact specificities of the Power architecture, we had to test our machines as black boxes, and modify our preliminary model in consequence. However, both the black box and adaptive model checking suppose deterministic systems, which is not our case.

Finally, I would like to explore the formalisation of diy by reusing ideas from D. Longuet *et al.* [LAG09]: they are able to generate tests from axiomatic specifications. Since we generate tests from violations of SC, which can be expressed as the negation of a simple property, I believe that this approach could be promising.

14.3 Other Models And Paradigms

I also would like to extend my framework to the study of high-level memory models, such as C++. I believe that it is more complicated to deal with software memory models than hardware ones. However, several ideas and principles seem to be roughly the same: therefore I believe that we could provide some insights on software models as well.

Moreover, this approach could provide a common reasoning framework for high- and low-level memory models, which I believe is lacking. Indeed, a memory model is meant to be a contract between a low-level layer and a higher one. Hence, it should be easy to relate high- and low-level models, but it is not, whereas the design of high-level models, *e.g.* C++ [BA], depends on a good comprehension of the underlying low-level models.

Finally, I also would like to see to what extent other paradigms, such as *transactional memory* or *message passing*, could also be described in terms that are common to weak memory models. Hence we could examine the relations (*e.g.* trade-offs or equivalences) between weak memory models as we studied and such paradigms.

14.4 Testing Semantics For Weak Memory Models

I believe that an interesting perspective could be the study of program transformations in the context of weak memory models, for example transformations induced by compilers.

The correction of program transformations has already been studied in the context of high-level memory models [Sev08], and recently for low-level ones, with a limited store atomicity relaxation [BMS]. These previous works examine the correctness of a transformation in a given model. We believe that our work could lead to proofs of transformations from one model to another, via *backward compatibility* proofs, such as our stability ones.

Testing semantics could be an appropriate formalism to study such transformations. In such a context, we could define the *behaviour* of a program as the set of its answers to a given *observation*, *i.e.* a set of tests.

It could lead to results on equivalence of programs, defining two programs to be equivalent when their behaviours—*w.r.t.* a given observation—are the same. We would define a transformation to be *safe w.r.t.* a given observation, *e.g.* an architecture, when it does not increase the behaviour of a program *w.r.t.* that observation. We could study such equivalences in a given model, for example trying to decide whether two PowerPC programs are equivalent, or if one implements the other safely.

But we could also study such equivalences from one memory model to another: does a certain PowerPC read-modify-write implements x86's `INC` instruction and conversely? Taking such a reasoning a step forward, we could also try for example to determine when it is safe to compile a given program towards a lock-free implementation.

14.5 Logics For Weak Memory Models

Another interesting perspective could be the design of a Hoare logic to study weak memory models. Logics to reason about concurrent programs already exist, *e.g.* Concurrent Separation Logic (CSL) [O'H]. However, they only address a restricted class of programs, *e.g.* the data-race free ones for CSL. We would like to study the design of a logic inspired of CSL but addressing racy programs as well. The main concern that I see yet is the meaning of the $\star$ operator in the context of racy programs. One interesting lead could be trying to restore the separation induced by $\star$ by using enough synchronisation.

The development of a target for certified compilers, such as X. Leroy's Compcert [Ler], or A. Appel and A. Hobor's Concurrent C Minor compiler [HAZN] could be a neat application of such a logic. Indeed these two compilers assume SC as the execution model of their target languages. We know that it is a safe assumption for data-race free programs, thanks to the DRF guarantee [FFS]. However, that does not allow us to reason about racy programs.

14.6 Partial Orders As a Model of Concurrency

Our framework could be the target of a soundness proof for this logic, inspired of S. Brookes' for CSL [Bro]. We believe our results on stability (see Chap. 11) could help design such a proof. S. Brookes defines in his CSL soundness proof a way to interleave the traces of two distinct parts of a program to build its executions. I believe that it will be very interesting to find a way to compose two of my orders to build an execution of a program from its distinct components. Moreover, this raises the question of *fairness* in weak memory models, *i.e.* if the way we compose those orders guarantees some fairness, or progress, in the execution of a program.

Bibliography

[AAS03] A. Adir, H. Attiya, and G. Shurek. Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. In *TPDS*, 2003.

[AB] S.V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. To appear in CACM.

[ABBM] M. F. Atig, A. Bouajjani, S. Burckhardt, and M. Musuvathi. On the verification problem for weak memory models. In *POPL 2010*.

[ABJ$^+$] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *SPAA 1993*.

[Adv93] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, December 1993.

[AFI$^+$] J. Alglave, A. Fox, S. Ishtiaq, M. O. Myreen, S. Sarkar, P. Sewell, and F. Zappa Nardelli. The Semantics of Power and ARM Multiprocessor Machine Code. In *DAMP 2009*.

[AG95] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer*, 29:66–76, 1995.

[AH] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *ISCA 1990*.

[alp02] Alpha Architecture Reference Manual, Fourth Edition, 2002.

[AM] Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. In *ISCA 2006*.

[AMSS] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in Weak Memory Models. In *CAV 2010*.

[APR99] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent Advances In Memory Consistency Models For Hardware Shared Memory Systems. *Proceedings of the IEEE*, 87(3):445–455, 1999.

[ARM08a] ARM. ARM Barrier Litmus Tests and Cookbook, February 2008.

[ARM08b] ARM Architecture Reference Manual (ARMv7-A and ARMv7-R), April 2008.

[ASa] D. Aspinall and J. Sevcik. Formalizing Java's data race free guarantee. In *TPHOL 2007*.

[ASb] D. Aspinall and J. Sevcik. Java Memory Model Examples: Good, Bad and Ugly.

[AS02] A. Adir and G. Shurek. Generating Concurrent Test-Programs with Collisions for Multi-Processor Verification. In *HLDVT*, 2002.

[BA] H.-J. Boehm and S.V. Adve. Foundations of the C++ Concurrency Memory Model. In *PLDI 2008*.

[BAM] S. Burckhardt, R. Alur, and M. K. Martin. Checkfence: Checking Consistency of Concurrent Data Types on Relaxed Memory Models. In *PLDI 2007*.

[BC] Y. Bertot and P. Casteran. *Coq'Art*. Springer Verlag, EATCS Texts in Theoretical Computer Science.

[BMa] S. Burckhardt and M. Musuvathi. Effective Program Verification for Relaxed Memory Models. In *CAV 2008*.

[BMb] S. Burckhardt and M. Musuvathi. Memory Model Safety of Programs. In *EC^2 2008*.

[BMS] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying Local Transformations of Concurrent Programs. In *CC 2010*.

[Boea] H.-J. Boehm. Reordering Constraints for Pthread-Style Locks. In *PPoPP 2007*.

[Boeb] H.-J. Boehm. Threads Cannot Be Implemented As a Library. In *PLDI 2005*.

[Boe07] H.-J. Boehm. Alternatives to SC. Message to the C++ standardisation list, January 2007. `http://www.decadentplace.org.uk/pipermail/cpp-threads/2007-January/001312.html`.

[BPa] G. Boudol and G. Petri. Relaxed Memory Models: An Operational Approach. In *POPL 2009*.

[BPb] G. Boudol and G. Petri. A Theory of Speculative Computation. In *ESOP 2010*.

[Bro] S. Brookes. A Semantics for Concurrent Separation Logic. In *TCS 2007*.

[CI] N. Chong and S. Ishtiaq. Reasoning About the ARM Weakly Consistent Memory Model. In *MSPC 2008*.

[CKS] P. Cenciarelli, A. Knapp, and E. Sibilio. The Java Memory Model: Operationally, Denotationally, Axiomatically. In *ESOP 2007*.

[CLS] J. Cantin, M. Lipasti, and J. Smith. The Complexity of Verifying Memory Coherence. In *SPAA 2003*.

[Col92] W. W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, 1992.

[CSB] F. Corella, J. M. Stone, and C. M. Barton. A Formal Specification of the PowerPC Shared Memory Architecture. Technical Report RC18638, IBM.

[DS90] M. Dubois and C. Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Transactions on Software Engineering*, 16(6), June 1990.

[FFS] R. Ferreira, X. Feng, and Z. Shao. Parameterized Memory Models and Concurrent Separation Logic. In *ESOP 2010*.

[FH] K. Fraser and T. L. Harris. Concurrent Programming Without Locks.

[FL] M. Frigo and V. Luchangco. Computation-Centric Memory Models. In *SPAA 1998*.

[Fri97] M. Frigo. The Weakest Reasonable Memory Model. Master's thesis, MIT, October 1997.

[Gha95] K. Gharachorloo. Memory Consistency Models for Shared-Memory Multiprocessors. *WRL Research Report*, 95(9), 1995.

[GK97] P. B. Gibbons and E. Korach. Testing Shared Memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[GLL$^+$] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA 1990*.

[GPY06] A. Groce, D. Peled, and M. Yannakakis. Adaptive Model Checking. 2006.

[HAZN] A. Hobor, A. Appel, and F. Zappa Nardelli. Oracle Semantics For Concurrent Separation Logic. In *ESOP 2008*.

[Her] M. Herlihy. Wait-Free Synchronisation. In *TOPLAS 1991*.

[Hil98] M. Hill. Multiprocessors Should Support Simple Memory Consistency Models. In *IEEE Computer*, 1998.

[HJK07] L. Higham, L. Jackson, and J. Kawash. What is Itanium Memory Consistency From the Programmer's Point of View? *Electronic Notes in Theoretical Computer Science*, 174(9):63–84, 2007. Proceedings of the Thread Verification Workshop (TV 2006).

[HKV98] L. Higham, J. Kawash, and N. Verwaal. Weak Memory Consistency Models Part i: Definitions and Comparisons. *Technical Report98/612/03, Department of Computer Science, The University of Calgary, January*, 1998.

[HP] M. Huisman and G. Petri. The Java Memory Model: A Formal Explanation.

[HS08] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan and Kaufmann, Burlington, 2008.

[HVM$^+$] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. TSOTool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In *ISCA 2004*.

[IBM02] IBM. *Book E - Enhanced PowerPC Architecture*. May 2002.

[int07] Intel 64 Architecture Memory Ordering White Paper, August 2007.

[int09] Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 3A, rev. 30, March 2009.

[ita02] A Formal Specification of Intel Itanium Processor Family Memory Ordering. October 2002. Intel Document 251429-001.

[LAG09] D. Longuet, M. Aiguier, and P. Le Gall. Proof-Guided Test Selection From First-Order Specifications With Equality. In *Journal of Automated Reasoning*, 2009.

[Lam79] L. Lamport. How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. *IEEE Trans. Comput.*, 46(7):779–782, 1979.

[Lea06] D. Lea. The JSR-133 Cookbook for Compiler Writers. September 2006.

[Lea07] D. Lea. Alternatives to SC. Message to the C++ standardisation list, January 2007. `http://www.decadentplace.org.uk/pipermail/cpp-threads/2007-January/001287.html`.

[Ler] X. Leroy. Formal Certification of a Compiler Back-End, or: Programming a Compiler With a Proof Assistant. In *POPL 2006*.

[LHF05] M. Lyons, B. Hay, and B. Frey. PowerPC Storage Model and AIX Programming, November 2005.

[LHH91] A. Landin, E. Hagersten, and S. Haridi. Race-Free Interconnection Networks and Multiprocessor Consistency. *SIGARCH Comput. Archit. News*, 19(3):106–115, 1991.

[LM67a] John Lennon and Paul McCartney. Magical mystery tour. In *Magical Mystery Tour*, 1967.

[LM67b] John Lennon and Paul McCartney. Sgt. peppers lonely hearts club band (reprise). In *Sgt. Peppers Lonely Hearts Club Band*, 1967.

[LM67c] John Lennon and Paul McCartney. With a little help from my friends. In *Sgt. Peppers Lonely Hearts Club Band*, 1967.

[mac] `http://www.hpcx.ac.uk/` and `http://www.idris.fr/`.

[MHAM] S. Mador-Haim, R. Alur, and M. K. Martin. Generating Litmus Tests For Contrasting Memory Consistency Models. In *CAV 2010*.

[MPA] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *POPL 2005*.

[O'H] P. W. O'Hearn. Resources, Concurrency, and Local Reasoning. In *TCS 2007*.

[OSS] S. Owens, S. Sarkar, and P. Sewell. A Better x86 Memory Model: x86-TSO. In *TPHOL 2009*.

[pow09] *Power ISA Version 2.06*. 2009.

[ppc] *PowerPC Implementation Features - Book IV*.

[ppc07] *Power ISA Version 2.05.* October 2007. `http://www.power.org/resources/reading/PowerISA_V2.05.pdf`.

[PVY99] D. Peled, M. Vardi, and M. Yannakakis. Black Box Checking. In *PSTV XIX*, 1999.

[SD87] C. Scheurich and M. Dubois. Correct Memory Operation of Cache-Based Multiprocessors. In *ISCA*, 1987.

[Sev08] J. Sevcik. *Program Transformations in Weak Memory Models.* PhD thesis, University of Edinburgh, 2008.

[SF95] J. M. Stone and R. P. Fitzgerald. Storage in the PowerPC. 1995.

[SJMvP] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A Theory of Memory Models. In *PPoPP 2007.*

[spa92] Sparc Architecture Manual Version 8, 1992.

[spa94a] Sparc Architecture Manual Versions 8 and 9, 1992 and 1994.

[spa94b] Sparc Architecture Manual Version 9, 1994.

[SS] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. In *TOPLAS 1988.*

[SSZN$^+$] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The Semantics of x86-CC Multiprocessor Machine Code. In *POPL 2009.*

[WSMF03a] J. Wetzel, E. Silha, C. May, and B. Frey. *PowerPC Operating Environment Architecture - Book III - Version 2.01.* December 2003.

[WSMF03b] J. Wetzel, E. Silha, C. May, and B. Frey. *PowerPC Virtual Environment Architecture - Book II - Version 2.01.* December 2003.

[WSMF03c] J. Wetzel, E. Silha, C. May, and B. Frey. *PowerPC User Instruction Set Architecture - Book I - Version 2.01.* September 2003.

[YGL] Y. Yang, G. Gopalakrishnan, and G. Lindstrom. UMM: an Operational Memory Model Specification Framework with Integrated Model Checking Capability. In *CCPE 2007.*

[YGLS03] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Analyzing the Intel Itanium Memory Ordering Rules Using Logic Programming And SAT. pages 81–95, 2003.

[YGLS04] Y. Yang, G. Gopalakrishnan, G. Linstrom, and K. Slind. Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models. *IPDPS*, 2004.

Part VI

Appendix

Appendix A

Uniprocessor Equivalences

We detail here the proofs of equivalence of the three formulations of the uniproc check we gave in Sec. 3.4.1.

We show easily that the transitive closure of $\xrightarrow{\text{com}}$, written $(\xrightarrow{\text{com}})^+$ is equal to $(\xrightarrow{\text{com}} \cup (\xrightarrow{\text{ws}};\xrightarrow{\text{rf}}) \cup (\xrightarrow{\text{fr}};\xrightarrow{\text{rf}}))$.

We write $x \overset{\text{hat}}{\leftrightarrow} y$ for $\exists w, w \xrightarrow{\text{rf}} x \wedge w \xrightarrow{\text{rf}} y$, *i.e.* when x and y are both reads, reading from the same write.

Let us consider these three formulations of uniproc:

$$\begin{array}{ll} (\mathit{Uni1}) & \mathsf{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}}) \\ (\mathit{Uni2}) & \forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow \neg(y\,(\xrightarrow{\text{com}})^+\,x) \\ (\mathit{Uni3}) & \xrightarrow{\text{po-loc}} \subseteq ((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow}) \end{array}$$

A.1 Some Handy Lemmas

We first show a few handy lemmas. We omit their proofs since they are easy:

Lemma 21 $(\xrightarrow{\text{com}})^+$ *is transitive*

Lemma 22 $(\xrightarrow{\text{com}})^+$ *is irreflexive*

Lemma 23 $(\xrightarrow{\text{com}})^+$ *is acyclic*

Lemma 24 $\forall xy, x \in \mathbb{R} \wedge y \in \mathbb{R} \wedge x\,(\xrightarrow{\text{com}})^+\,y \Rightarrow x(\xrightarrow{\text{fr}};\xrightarrow{\text{rf}})y$

Lemma 25 $((\xrightarrow{\text{com}})^+;\overset{\text{hat}}{\leftrightarrow}) \subseteq (\xrightarrow{\text{com}})^+$

Proof *Suppose $x\,(\xrightarrow{\text{com}})^+\,r \overset{\text{hat}}{\leftrightarrow} y$ and do a disjunction on the direction of x.*

- *$x \in \mathbb{R}$: by Lem. 24 we know $x(\xrightarrow{\text{fr}};\xrightarrow{\text{rf}})r$. Let us write w_r for the rf write of r, we have by hat: $w_r \xrightarrow{\text{rf}} y$. Thus $x \xrightarrow{\text{fr}} w_r \xrightarrow{\text{rf}} y$, hence $x\,(\xrightarrow{\text{com}})^+\,y$.*

- $x \in \mathbb{W}$: *let us write* w_r *for the rf write of* y*. If* $x = w_r$*, then* $x \xrightarrow{\text{rf}} y$ *by hat. Otherwise, either* $x \xrightarrow{\text{ws}} w_r$*, thus* $x(\xrightarrow{\text{ws}};\xrightarrow{\text{rf}})y$*, or* $w_r \xrightarrow{\text{ws}} x$*, thus* $r \xrightarrow{\text{fr}} x$*, which leads to a cycle in* $(\xrightarrow{\text{com}})^+$*, a contradiction of Lem. 23.* □

Lemma 26 $\exists x, x\,(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}})^+\, x \Rightarrow \exists y, y\,((\xrightarrow{\text{com}})^+; \xrightarrow{\text{po-loc}})^+\, y$

Proof *Since* $\xrightarrow{\text{com}} \subseteq (\xrightarrow{\text{com}})^+$*, and* $(\xrightarrow{\text{com}})^+$ *and* $\xrightarrow{\text{po-loc}}$ *are transitive and irreflexive, Lem. 1 applies.*

A.2 (*Uni2*) and (*Uni3*) Are Equivalent

Lemma 27 (*Uni3*) $\Rightarrow$ (*Uni2*)

Proof *Suppose* (*Uni3*)*, that is:* $\forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow x((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow})y$*. We have to prove:* $\neg(y\,(\xrightarrow{\text{com}})^+\, x)$*. Let us suppose as a contradiction* $y\,(\xrightarrow{\text{com}})^+\, x$*. Let us do a case disjunction over* $x((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow})y$*:*

- *if* $x\,(\xrightarrow{\text{com}})^+\, y$*, we create a cycle in* $(\xrightarrow{\text{com}})^+$*, thus a contradiction of Lem. 23.*
- *if* $x \overset{\text{hat}}{\leftrightarrow} y$*: there is* w *s.t.* $w \xrightarrow{\text{rf}} x$ *and* $w \xrightarrow{\text{rf}} y$ *by hat. As* x *and* y *are both reads and* $y\,(\xrightarrow{\text{com}})^+\, x$*, we know by Lem. 24 that:* $y(\xrightarrow{\text{fr}};\xrightarrow{\text{rf}})x$*. As the rf write of* x *is unique, we actually have:* $y \xrightarrow{\text{fr}} w \xrightarrow{\text{rf}} x$ *thus* $y \xrightarrow{\text{fr}} w$ *and* $w \xrightarrow{\text{rf}} y$*, which leads to a cycle in* $(\xrightarrow{\text{com}})^+$*, a contradiction of Lem. 23.* □

Lemma 28 (*Uni2*) $\Rightarrow$ (*Uni3*)

Proof *Suppose* (*Uni2*)*, that is:* $\forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow \neg(y\,(\xrightarrow{\text{com}})^+\, x)$*. We have to prove:* $x((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow})y$*. Let us do a case disjunction over the directions of* x *and* y*:*

- $(x, y) \in \mathbb{R} \times \mathbb{R}$*: let us write* w_x *and* w_y *the rf writes for* x *and* y*. If* $w_x = w_y$*, we have* $x \overset{\text{hat}}{\leftrightarrow} y$*. If not, either* $w_x \xrightarrow{\text{ws}} w_y$ *or* $w_y \xrightarrow{\text{ws}} w_x$*. If* $w_x \xrightarrow{\text{ws}} w_y$*, then* $x \xrightarrow{\text{fr}} w_y$*, thus:* $x(\xrightarrow{\text{fr}};\xrightarrow{\text{rf}})y$*. If* $w_y \xrightarrow{\text{ws}} w_x$*, then* $y \xrightarrow{\text{fr}} w_x$*, thus* $y(\xrightarrow{\text{fr}};\xrightarrow{\text{rf}})x$ *which contradicts* (*Uni2*)*.*
- $(x, y) \in \mathbb{W} \times \mathbb{R}$*: let us write* w_y *the rf write for* y*. If* $x = w_y$*, then* $x \xrightarrow{\text{rf}} y$*. If not, then either* $x \xrightarrow{\text{ws}} w_y$ *or* $w_y \xrightarrow{\text{ws}} x$*. If* $x \xrightarrow{\text{ws}} w_y$*, then* $x(\xrightarrow{\text{ws}};\xrightarrow{\text{rf}})y$*. If* $w_y \xrightarrow{\text{ws}} x$*, then* $y \xrightarrow{\text{fr}} x$*, which contradicts* (*Uni2*)*.*
- $(x, y) \in \mathbb{R} \times \mathbb{W}$*: symmetric of* $\mathbb{W} \times \mathbb{R}$ *case.*
- $(x, y) \in \mathbb{W} \times \mathbb{W}$*: we know* $x \neq y$ *as* $x \xrightarrow{\text{po-loc}} y$*. Thus either* $x \xrightarrow{\text{ws}} y$*, which leads to the result, or* $y \xrightarrow{\text{ws}} x$ *which contradicts* (*Uni2*)*.* □

A.3 (*Uni1*) and (*Uni2*) Are Equivalent

Lemma 29 (*Uni1*) $\Rightarrow$ (*Uni2*)

Proof *Suppose* (*Uni1*)*, i.e.* $\mathsf{acyclic}(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}})$*. We want:* $\forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow \neg(y\,(\xrightarrow{\text{com}})^+\, x)$*. Suppose* $x \xrightarrow{\text{po-loc}} y$ *and* $y\,(\xrightarrow{\text{com}})^+\, x$*. By induction on* $y\,(\xrightarrow{\text{com}})^+\, x$*, we build a cycle in* $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}}$*, contradicting* (*Uni1*)*.* □

Lemma 30 *(Uni2)* $\Rightarrow ((\xrightarrow{\text{com}})^+ \cup \xrightarrow{\text{po-loc}}) \subseteq (\xrightarrow{\text{com}})^+$

Proof *Let us suppose* $x((\xrightarrow{\text{com}})^+; \xrightarrow{\text{po-loc}})y$, *i.e.* $x\,(\xrightarrow{\text{com}})^+\, z \xrightarrow{\text{po-loc}} y$. *We have* $(Uni2)$ *as an hypothesis; moreover we know that* $(Uni2)$ *implies* $(Uni3)$, *thus* $z((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow})y$. *Therefore:* $x((\xrightarrow{\text{com}})^+; ((\xrightarrow{\text{com}})^+ \cup \overset{\text{hat}}{\leftrightarrow}))y$, *i.e.* $x(((\xrightarrow{\text{com}})^+; (\xrightarrow{\text{com}})^+) \cup ((\xrightarrow{\text{com}})^+; \overset{\text{hat}}{\leftrightarrow}))y$. *By Lem. 21, we have* $((\xrightarrow{\text{com}})^+; (\xrightarrow{\text{com}})^+) \subseteq (\xrightarrow{\text{com}})^+$. *By Lem. 25, we have* $((\xrightarrow{\text{com}})^+; \overset{\text{hat}}{\leftrightarrow})$ *included in* $(\xrightarrow{\text{com}})^+$. *Thus* $x\,(\xrightarrow{\text{com}})^+\, y$.

Lemma 31 $(Uni2) \Rightarrow (Uni1)$

Proof *Suppose* $(Uni2)$, *i.e.:* $\forall xy, x \xrightarrow{\text{po-loc}} y \Rightarrow \neg(y\,(\xrightarrow{\text{com}})^+\, x)$. *We have to prove that there is no cycle in* $\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}}$. *Suppose there is one, ie* $\exists a, a\,(\xrightarrow{\text{com}} \cup \xrightarrow{\text{po-loc}})^+\, a$. *Thus by Lem. 26 we have:* $\exists e, e\,((\xrightarrow{\text{com}})^+; \xrightarrow{\text{po-loc}})^+\, e$. *By Lem. 30, we know that* $e\,((\xrightarrow{\text{com}})^+)^+\, e$, *i.e.* $e\,(\xrightarrow{\text{com}})^+\, e$, *a contradiction of Lem. 23.* □

Appendix B

A Word on the Coq Development

We give here some insights on the Coq development.

B.1 Overview of the Development

We give in Fig. B.1 a graphic representation of the development, and in Fig. B.2 the numbers of specification and proof lines per module. The whole development counts about 24000 lines of code. It is available at `http://moscova.inria.fr/~alglave/wmm`.

The module `util` contains the definitions relative to relations and orders, and the associated lemmas, in particular the key lemma presented in Chap. 2.

The module `wmm` contains the definitions relative to the objects of our model, as described in Sec. 3.1. It contains the definition of an architecture as a `Module Type` and of a weak memory model as a `Module`.

The module `basic` contains basic lemmas over the objects of our framework. The module `uniproc` contains the proofs of equivalence of the three formulations of the `uniproc` check, as described in App. A.

The `hierarchy` module contains the definitions of the weaker predicate (see Sec. 3.5), and the theorems relative to validity presented in Sec. 1 and Sec. 2. The module `valid` contains the proof presented in Sec. 8, that any order compatible with $\xrightarrow{\text{ppo}}$ and satisfying `uniproc` builds a valid execution. The modules `sc`, `tso`, `pso`, `rmo` and `alpha` contain the definitions and proofs of equivalence relative to the eponymous models.

The `stable` module contains the definitions of covering and well-founded synchronisation relation, and the associated lemmas presented in Chap. 11. The modules `drf` and `racy` contain an instantiation of `stable` recovering respectively the DRF guarantee of Sec. 10.2 and the lock-free guarantee of Sec. 10.3. The `shasha` module contains our study of critical cycles, presented in Chap. 11. The modules `locks` and `rmw` contain the lock-based (Sec. 10.4.2) and lock-free synchronisation (see Sec. 10.4.3) studies.

B.2 Basic Objects

We describe here how we represent the basic objects presented in Sec. 3.1.

B.2.1 Basic Types

We use `set` as a shortcut for `Ensemble`, *i.e.* `fun U : Type -> U -> Prop`. We define the type `Rln` of our relations as:

```
Definition Rln (A:Type) := A -> A -> Prop.
```

See the module `util` for more details.

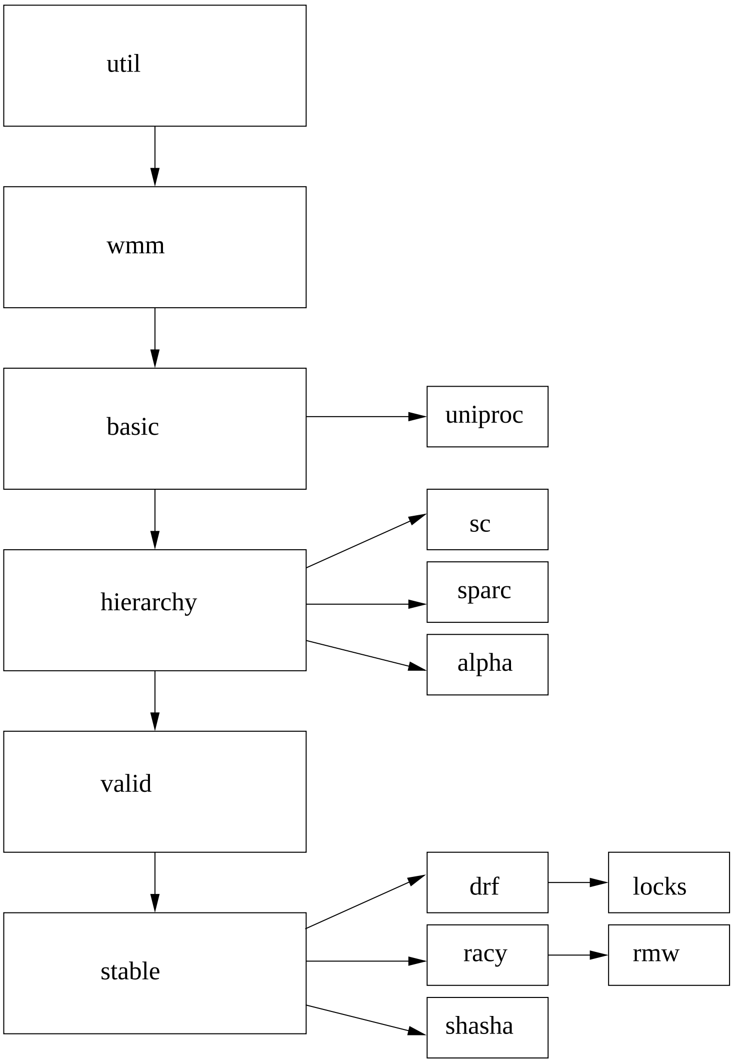


Figure B.1: Overview Of The Development

Module	Specification	Proof	Comments
util	466	1085	95
wmm	250	24	115
basic	856	2319	55
hierarchy	336	1182	22
uniproc	163	762	21
sc	330	672	34
tso	643	1892	57
pso	368	878	30
rmo	685	2022	48
alpha	573	1958	45
valid	260	599	30
stable	157	292	19
drf	166	174	3
racy	179	316	9
shasha	617	1111	14
locks	412	817	65
rmw	176	331	10
total	6637	16434	672

Figure B.2: Numbers Of Lines Per Module

B.2.2 Events and Program Order

We consider a `Word` to be a `nat`; an `Address` and a `Value` are both of type `Word`. We describe a processor by a type `Proc` which is `nat`. Similarly, the *program order index* (*i.e.* an index indicating when the event was issued in the program order) is described by a type `Poi` which is `nat`. An *instruction identifier* of type `Iiid` gives the processor and the program order index of an event.

An event is described as a `Record`, given in Fig. B.3. It is composed of an `Eiid` (which is a unique identifier of type `nat`), an `Iiid` (which gives the processor and the program order index of the event), and an `Action`.

The type `Dirn` of *directions* is an inductive with two cases, one for read, the other for write. An `Action` is an inductive with one case, composed of a `Dirn`, a `Location` and a `Value`, as depicted in Fig. B.3.

We describe the program order as a relation over a set of events, given in Fig. B.4. Two events are in program order if the processors given by their `Iiid` are equal, and if the `Poi` of the first is less than or equal to the `Poi` of the second. See the module `wmm` for more details.

```
(*Proc, Poi, Location, Value, Eiid are equal to nat*)
Record Iiid  : Set := mkiiid {
  proc : Proc;
  poi: Poi}.
Inductive Dirn : Set :=
  | R : Dirn
  | W : Dirn.
Inductive Action : Set :=
  | Access : Dirn -> Location -> Value -> Action.
Record Event : Set := mkev {
  eiid : Eiid;
  iiid : Iiid;
  action : Action}.
```

Figure B.3: Code For `Event` Type

```
Definition po (es: set Event) : set (Event*Event) :=
  fun c => match c with (e1,e2) =>
    (*both events have same processor*)
      (e1.(iiid).(proc) = e2.(iiid).(proc)) /\
    (*the program order index of e1 is less than e2's*)
      (le e1.(iiid).(poi) e2.(iiid).(poi)) /\
    (*both e1 and e2 are in the set of events*)
      (In _ es e1) /\ (In _ es e2)
  end.
```

Figure B.4: Program Order Code

B.2.3 Execution Witnesses

We define an *event structure* as a `Record` composed of two fields, the first one being the set of events, the second the *intra-instruction causality*, if any, written `iico`:

```
Record Event_struct : Type := mkes {
  events : set Event;
  iico : Rln Event}.
```

We define the union of `po` and `iico`, written `po_iico`.

We define the read-from map, the write serialisation and the from-read map as `Rln` over `Event`. We define the associated well-formedness predicates following the definition given in Sec. 3.2.1 and 3.2.2.

An execution witness is described as a `Record` with two fields, one for the read-from map, one for the write serialisation:

```
(*Write_serialisation and Rfmap are Rln Event*)
Record Execution_witness : Type := mkew {
  ws : Write_serialisation;
  rf : Rfmap}.
```

See the module `wmm` for more details.

B.3 Architectures and Weak Memory Models

An architecture is a module following the module type given in Fig. B.5. Given an event structure and an associated execution witness, a module of type `Archi` produces:

- a preserved program order `ppo` as described in Sec. 3.3.2,
- the global read-from map `grf` as described in Sec. 3.3.1,
- the ordering induced by barriers `ab` as described in Sec. 3.3.3 and Chap. 8, and
- the `stars` events for synchronisation, as described in Chap. 10.

A weak memory model is a module which takes two arguments. The first one is an architecture of type `Archi`. The second one is the $\xrightarrow{\text{dp}}$ relation, defined as a module type following Sec. 3.4.1.3: this module constructs a relation over events, included in the program order, transitive, and with a read as its source. The code of that module is given at Fig. B.6.

A weak memory model will produce, given an architecture and a dependency relation:

- a global happens-before relation, following the definition in Sec. 3.3.4,

```
Module Type Archi.
Parameter ppo : Event_struct -> Rln Event.
Hypothesis ppo_valid : forall (E:Event_struct),
  rel_incl (ppo E) (po_iico E).

Parameter grf : Execution_witness -> Rln Event.
Hypothesis grf_valid : forall (X:Execution_witness),
  rel_incl (grf X) (rf X).

Parameter ab : Event_struct -> Execution_witness -> Rln Event.
Hypothesis ab_evts :
  forall (E:Event_struct) (X:Execution_witness),
    forall x y, well_formed_event_structure E ->
    ab E X x y -> In _ (events E) x /\ In _ (events E) y.

Parameter stars : Event_struct -> set Event.
End Archi.
```

Figure B.5: Module Type For Architectures

```
Module Type Dp.
Parameter dp : Event_struct -> Rln Event.
Hypothesis dp_valid : forall (E:Event_struct),
  rel_incl (dp E) (po_iico E) /\ trans (dp E) /\
  forall x y, dp E x y -> reads E x.
End Dp.
```

Figure B.6: Code For The Dp Module

```
Module Wmm (A : Archi) (dp : Dp).
Import A.
Import dp.

Definition ghb (E:Event_struct) (X:Execution_witness) :
  Rln Event :=
    rel_union (grf X) (rel_union (ab E X)
      (rel_union (rel_union (ws X) (fr E X)) (ppo E))).

Definition valid_execution
  (E:Event_struct) (X:Execution_witness) : Prop :=
  write_serialisation_well_formed (events E) (ws X) /\
  rfmaps_well_formed (events E) (rf X) /\
  acyclic (rel_union (com E X) (po-loc-llh E)) /\ (*uniproc*)
  acyclic (rel_union (rf X) (dp E)) /\ (*thin*)
  acyclic (ghb E X).
End Wmm.
```

Figure B.7: Code For Wmm Module

- a validity check for any execution, following the definition in Sec. 3.4.

For the uniproc check (see Sec. 3.4.1), we define the load-load hasard $\xrightarrow{\text{po-loc}}$ relation as follows, where `loc` extracts the location of an event and `reads E` is the subset of `E` which are reads, *i.e.* having `R` as direction:

```
Definition pio_llh (E:Event_struct) : Rln Event :=
  fun e1 => fun e2 =>
    loc e1 = loc e2 /\ po_iico E e1 e2 /\
    ~(reads E e1 /\ reads E e2).
```

We give in Fig. B.7 the code for the module `Wmm`. See the module `wmm` for more details.

B.4 Proofs

There is not much to say about the proofs themselves. The code is not very smart, and our proof techniques are standard from a paper proof point of view. I most of the time use proofs by contradiction, as one can see along this manuscript. One can read the proof of Lem. 1 to see the kind of reasoning that I use. I find it convenient to deal with the most common proof pattern that I met, *i.e.* "if there is a cycle in this relation, then there is a cycle in that other one, which is known to be acyclic by definition, hence a contradiction". One thing that can be noted is that I make a heavy use of excluded middle, added as an axiom to my development.

However, I believe that this is precisely because I did my proofs in Coq that I have been able to reduce the number of axioms involved in my framework (see Chap. 3). Similarly, the observations that an A-cumulative barrier is enough to restore SC (see Sec. 10.3.3), or that we do not need all the features of the PowerPC barriers to build correct locks (see Chap. 10), come from my formal development.

Index

Alpha
  Check, 66
  Implementation, 49, 66
  Native Definition, 66
  Preserved Program Order, 49, 66

Barriers
  A-Cumulative, 43
  B-Cumulative, 43
  Constraints, 42
  Dynamic Property, 135
  Non-Cumulative, 42
  Static Property, 135

Candidate Relaxations, 79
  Barriers
    Cumulativity Ordering, 83
    Definition, 86
    Plain Ordering, 82
  Coherence, 80
  Communication, 85
  Composite, 88
  Control Dependencies
    Definition, 88
    Load-Store Quotation, 82
  Data Dependencies
    Definition, 87
    Load-Load Quotation, 81
    Load-Store Quotation, 81
  Global, 79
  Non-Global, 79
  Program Order, 85
  Relaxed, 79
  Safe, 79
  Store Buffering, 80
Communication Relation, 39
Competing Accesses, 130
Configuration File, 95
Conflict Relation, 127
Covered
  Execution, 127
Covering Relation, 128
Critical
  Cycle, 153

Data-Race-Free
  Covered, 130
  Guarantee (Generalised), 131
Dependencies, 47

Events
  Barrier, 108
  Commit, 108
  Direction, 35
  Location, 35
  Memory, 35
  Processor, 35
  Register, 106
Execution Witness
  Definition, 40
  Well-Formedness, 40

Fragile
  Pairs, 132
  Reads, 133
From-Read Map, 39
Fully Barriered Execution, 134

Global Happens-Before, 40
  Definition, 44
Globally Performed, 40

Hat Relation, 46

Intra-Instruction Causality, 107

Litmus Tests, 92
Load-Load Hasard, 47
Lock-Based Synchronisation, 130
Lock-Free
- Covered, 133
- Guarantee, 137

Model
- Architecture, 43
- Thin Air Check, 49
- Uniprocessor Check, 45
- Validity of an Execution, 49
- Weaker Predicate, 50

Orders
- Linear Extension, 25
- Partial, 24
- Total or Linear Strict, 24

Preserved Program Order, 42
Program Order, 35

Read Others' Writes Early, 41
Read Own Writes Early, 41
Read-From Map
- Definition, 38
- Global, 42
- Internal and External, 41
- Well-Formedness, 38

Relations
- Acyclicity, 24
- Domain and Range, 23
- Hexa, 26
- Irreflexivity, 23
- Reflexive Closure, 26
- Sequence, 25
- Totality, 24
- Transitive Closure, 23
- Transitivity, 23

Sequential Consistency
- Check, 60
- Implementation, 44, 60
- Native Definition, 60

Sequential Execution Model, 80
Sparc
- PSO
  - Check, 63
  - Implementation, 63
  - Native Definition, 63
  - Preserved Program Order, 63
- RMO
  - Check, 65
  - Implementation, 47, 64
  - Native Definition, 64
  - Preserved Program Order, 47, 64
- TSO
  - Check, 62
  - Implementation, 62
  - Native Definition, 44, 62
  - Preserved Program Order, 44, 62
- Value Axiom, 61

Store Atomicity Relaxation, 41
Store Buffering, 41
Synchronisation Relation, 127

Well-Founded Synchronisation, 128
Write Serialisation
- Definition, 38
- Well-Formedness, 38